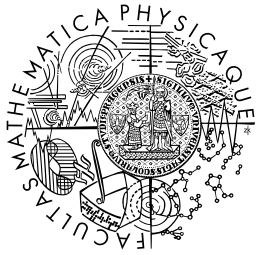

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Roman Firment

Monitoring Support for Manta Flow Agent in Cloud-Based Architecture

Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Pavel Parízek, Ph.D.
Study programme: Computer Science
Study branch: Software Systems

Prague 2022

I hereby declare that I have authored this thesis independently, and that all sources used are declared in accordance with the "Metodický pokyn o etické přípravě vysokoškolských závěrečných prací".

I acknowledge that my thesis (work) is subject to the rights and obligations arising from Act No. 121/2000 Coll., on Copyright and Rights Related to Copyright and on Amendments to Certain Laws (the Copyright Act), as amended, (hereinafter as the "Copyright Act"), in particular § 35, and § 60 of the Copyright Act governing the school work.

With respect to the computer programs that are part of my thesis (work) and with respect to all documentation related to the computer programs ("software"), I hereby grant the so-called MIT License.

The MIT License represents a license to use the software free of charge. I grant this license to every person interested in using the software. Each person is entitled to obtain a copy of the software (including the related documentation) without any limitation, and may, without limitation, use, copy, modify, merge, publish, distribute, sublicense and / or sell copies of the software, and allow any person to whom the software is further provided to exercise the aforementioned rights. Ways of using the software or the extent of this use are not limited in any way.

The person interested in using the software is obliged to attach the text of the license terms as follows:

In ..... date ....

Author's signature

I would like to thank my family for their full support throughout my university studies and for the encouragement, they were giving me.

A huge thanks to my thesis supervisor doc. RNDr. Pavel Parízek, Ph.D, for his patience and helpful guidance during the completion of my thesis. Then I would like to thank colleagues from the Utils Team and Manta Software, Inc., for the opportunity to work with them.

Title: Monitoring Support for Manta Flow Agent in Cloud-Based Architecture

Author: Roman Firment

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Today, it is quite common to see software migrations from an on-premises solution to a cloud solution. The product MANTA Flow Platform also experiences this transformation. As a part of this transformation, the introduction of a new component, the MANTA Flow Agent, is necessary. MANTA Flow Agent is a Java application supposed to run on a customer's machine and execute received commands from a remote running on a cloud side. This yields the natural need for the consideration of monitoring support. In this master thesis, we describe in more detail the responsibility of the MANTA Flow Agent, the multi-agent environment, motivation and requirements for the monitoring support for the MANTA Flow Agent. Furthermore, we provide an analysis of suitable technologies which could be used to bring such monitoring support. Then, we discuss different alternatives and solutions and their fit in our context. Finally, we describe the architecture for monitoring support and the implementation of a simple proof of concept solution based on a Java agent and the Prometheus, a time series database.

Keywords: monitoring, Java instrumentation agent, time series database, multi-agent environment, cloud architecture

Contents

1. Introduction

Today, it is very common that many companies have to deal with some sort of data. To work with data, different technologies could take part in it. This data could be used for different purposes and they could have different roles for such companies. However, at some point, it could be beneficial to have better insight into such data. What they look like, how their flow in a company system(s) or product looks like, what their origins are, how they transform across the data manipulation pipeline, and so on.

To resolve such issues, *data lineage* can be used. With data lineage, we can see the visualization of the way data flows within the system. We can get a map of data flows in a specific environment, a detailed description of data flows. It can be also used to track and explore the changes over time. Other use-cases are data auditing, data lineage as an additional element in approving the correctness of important decisions, helping with migration to different data technology, inspecting data incidents, and so on.

One of the software companies which offer a solution for generating and manipulating data lineage is Manta Software, Inc., Prague based company. Its core product is *MANTA Flow Platform*, an automated data lineage platform. It supports more than 17 databases technologies (PostgreSQL, Oracle, Google Big Query, ..), 4 programming languages (COBOL, Java, C#, Python), more than 8 reporting and analysis technologies (SAS, Power BI, OBIEE, ..). This set is still growing to cover the most used and popular technologies in the enterprise world where data is an inseparable part of it.

MANTA Flow Platform software is currently implemented as an *on-premises* solution. From the business perspective, it's necessary to make MANTA Flow Platform able to run in *cloud environments*. Such a cloud environment could be also a private cloud, but the long-term goal is to offer MANTA Flow Platform as a *SaaS* (Software as a service). MANTA Flow Platform in a cloud environment could benefit from large storage, high computing capabilities on demand, smaller maintenance costs, simplifying initial deployment, automatic updates, and from other properties of running software in the cloud.

To become more cloud-friendly, some architectural changes are required. One of them is splitting up some functionality into a separate component. This component is called *MANTA Flow Agent*. That component has a specific role and it is supposed to be installed within a customer's environment and will interact with the other core component, installed far away from the agent, in a cloud environment. As MANTA Flow Agent will be deployed outside of our environment, it's necessary to consider monitoring of such entities, to make their interactions flawless.

In this thesis, we mainly analyze requirements for monitoring support for MANTA Flow Agent, analyze existing monitoring solutions, design and provide an implementation for such support.

In the following subsections, we describe MANTA Flow Platform and MANTA Flow Agent in more detail, and we mention our motivation and specify the goals of this thesis within their context. Finally, the outline of this thesis can be found there.

1.1 MANTA Flow Platform

MANTA Flow Platform is a data lineage platform that can *scan data environment* to build a powerful map of all data flows and deliver it through a native UI and other channels to both technical and non-technical users.

The user just have to create a configuration to access a data source, e.g. a JDBC (Java Database Connectivity) URL with access credentials and define custom scanning process. The scanning process can include multiple data sources, data technologies, and consists of multiple phases. Some phases are optional, as we will see in the next section.

The result of such a scanning process is data lineage, data flow, which can be visualized by its visualization component. Scanned and analyzed metadata from data sources could be also integrated into some data catalog (a detailed inventory of all data assets in a data environment) in the optional export phase.

1.1.1 Scanning Process and Extraction Phase

Scan of a particular technical resource (such as a single database) is a process involving the capture, analysis, loading, and merging of lineage metadata into the MANTA Repository, metadata storage component. Scans are typically run against a set of connections for a given technology (such as all Oracle databases in a data environment). It consists of three phases: *extract*, *analyze* and *export* phase, as we can see in Figure 1.1.

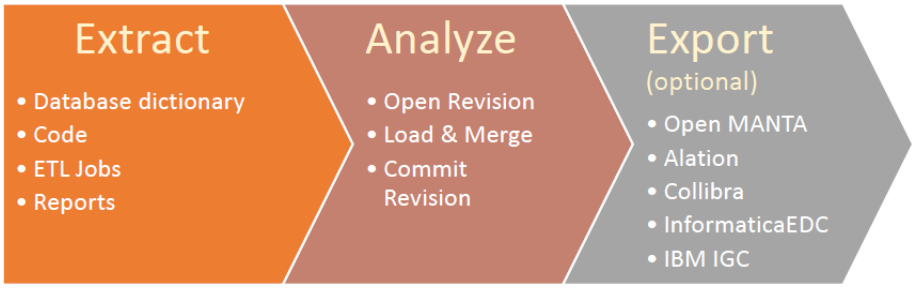

Figure 1.1: The Phases of MANTA Flow Scanning Process

We are interested only in the *extract* phase because this phase will run on the MANTA Flow Agent. During this phase, metadata is extracted from the data environment. Where possible, this is done via direct connection to the data source, typically to some DBMS, ETL (extract, transform and load) or reporting tools. This ensures that the metadata being retrieved is the most current and most relevant and reflects the truth of how the system is running today. This is usually done by APIs, but it depends on the technology. For most databases, this is done by directly accessing the database catalog. The extract phase first retrieves database dictionary information (primarily tables and columns) and then picks up assets that define lineage. These are typically views and stored procedures for a database but might be external data access steps and transformation modules when applied to an *ETL* tool. *Business intelligence* (reporting) solutions deliver lineage details about their queries along with columns in the report and their potential transformations. The extracted metadata is then ready for analysis.

As said before, the extraction logic from scanning process is now moved into the separate component, MANTA Flow Agent, where the extraction phase runs on.

1.1.2 MANTA Admin GUI

MANTA Admin GUI is MANTA Flow Platform utility web application that acts mainly as a graphical and a programming interface to do configuration and to launch data environment scanning. Within this application, users can configure data sources to be scanned and also define and execute complex workflows for scanning, which uses those data sources.

This component is the place, where you can manage MANTA Flow Agents such as registering them and choosing them as an executor for the extraction phase in the scanning process for some data sources, etc.

For our purpose, this component has a key role for us as it's the main component with which MANTA Flow Agent interacts. In the production, this is *the remote side* from the MANTA Flow Agent's perspective. It's also responsible for sending control commands for MANTA Flow Agent, as MANTA Flow Agent acts only as an executor for those commands.

1.2 MANTA Flow Agent and Multi-Agent Environment

The main motivation to have MANTA Flow Agent is to provide an option to get metadata from the data sources, which are not easily accessible from outside of the environment where data sources live. Such environments could be setups where access to data sources is behind the firewall or are accessible only from the local network. To overcome this, MANTA Flow Agent is a kind of insider, which is installed within such an environment, can access those data sources and that access to MANTA Admin GUI is under the control.

This component also opens a door to offering MANTA Flow Platform as SaaS, where customers can use service on demand, just by deploying MANTA Flow Agents in their systems, which have access to their data environment. They do not have to spend time on operational tasks related to MANTA Flow Platform.

The main role of MANTA Flow Agent is the execution of commands from MANTA Admin GUI. Those commands handle the extraction of metadata from data sources and also transporting extracted metadata to MANTA Admin GUI.

1.2.1 MANTA Flow Agent Properties

These are basic traits of MANTA Flow Agent which have been gathered during our initial analysis and we want to present them here to emphasize MANTA Flow Agent's important properties. MANTA Flow Agent:

- is a Java application, written in Spring Boot Framework. The same holds also for MANTA Admin GUI.

- its installation target platforms are Linux and Windows operating system families.
- to communicate with the MANTA Admin GUI the messaging is used, concretely the Java Message Service (JMS).
- doesn't interact with other agents. Interactions are allowed with MANTA Admin GUI, which serves as a control node, and with spawned extraction processs, as we will see in the next section.
- can connect but cannot be connected.
- is a long-run process, usually started in a daemon mode, which waits for commands from the MANTA Admin GUI.
- spawns extraction processes. Those extraction processes are again Spring Boot Java applications, which also communicate with MANTA Admin GUI using JMS. Those processes usually interact with data sources from which MANTA Flow Agent wants to extract metadata.
- as it spawns extraction processes, having installed multiple MANTA Flow Agent on the same host doesn't make much sense.
- its basic status (running/not running) for extraction processes is already implemented in MANTA Admin GUI.

1.2.2 Multi-Agent Environment

In the previous section, we also slightly mentioned the interactions between the MANTA Flow Agent and the environment where the agent could be installed.

In summary, the interaction between MANTA Flow Agent and the environment depends on which extraction processes it spawns. The minimal requirement is that MANTA Flow Agent can establish a connection to MANTA Admin GUI. We keep in mind, that MANTA Flow Agent could run within some container technologies, so MANTA Flow Agent or its spawn processes could observe on runtime more drastic resources changes than running on a native platform.

In Figure 1.2, we can see a basic communication scheme in a Multi-Agent environment. Squares *A1* and *A2* represent MANTA Flow Agents. In most scenarios, they are installed on different hosts. There are 2 demilitarized zones. There are both MANTA Flow Agents in the first zone, *DMZ1*. In the second demilitarized zone, *DMZ2*, only one MANTA Flow Agent is in. Being in a demilitarized zone, it makes possible for MANTA Flow Agent and its spawned extractor processes to access data source systems. Admin GUI is *the remote side* and there is communication between Admin GUI and MANTA Flow Agents and their spawn extractor processes *E1*, *E2* and *E3* with Admin GUI. JMS messaging is used for that communication.

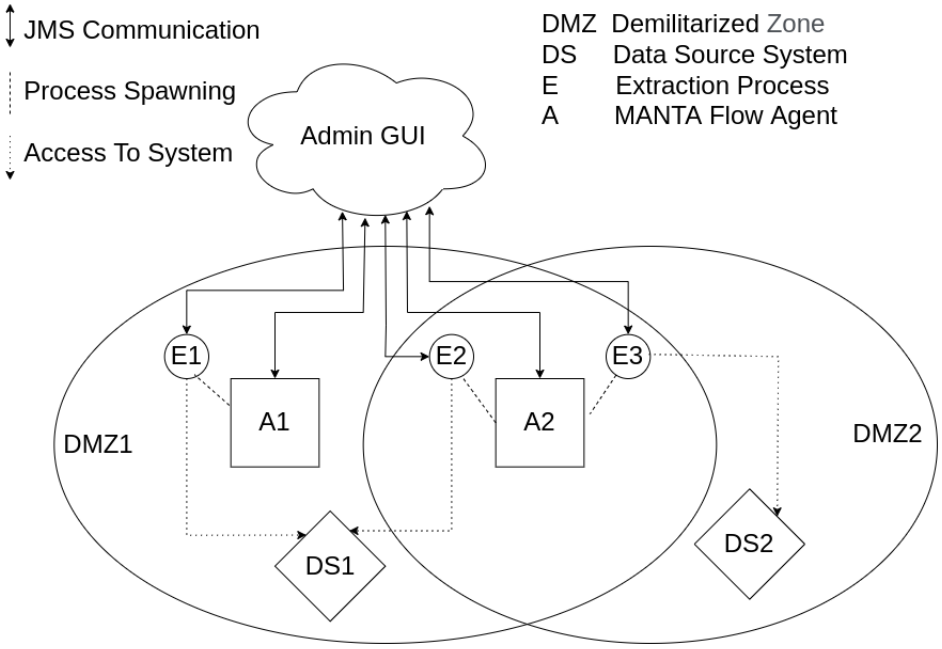

Figure 1.2: Communication Scheme in Multi-Agent Environment

1.3 Motivation For Monitoring Support

In this section, we would like to provide our motivation for considering and implementing monitoring support for MANTA Flow Agent in Cloud-Based Architecture.

The following point summaries our motivation to consider monitoring for MANTA Flow Agent:

- to provide basic information about MANTA Flow Agent resources utilization, runtime statistics from the current and previous workflows executions. Such workflows describe user-defined scanning processes. Some gathered and statistically computed metrics from monitoring components should be faced to the user in some visual form, e.g as an extension of already planned dashboards in Admin GUI. Those metrics could be e.g.: available processors, free/total RAM and used RAM by the agent, disk space, system and the agent CPU load, network and filesystem activity, how much resources were consumed by execution of one workflow, etc. The purpose is also to inform the user about static metrics from multiple agents and based on that the user could decide on which agent some extraction should be run. Another one is that this visual information can be used not only to inform the user about the current multi-agent system state but also could help with troubleshooting.

- to have a runtime anomalies detection mechanism. MANTA Flow Agent and its spawned extractor processes are processes that consume and use some resources. An anomalies detection mechanism could process output from the monitoring and warn the user about suspicious states or when some values overrun limits. Those deviations could be derived from the

historical data from previous runs and help a customer and us to identify the roots of some performance issues, e.g. on which side (on the customer's or MANTA's side) it happened. Some examples for those detections could be: memory leaks detection (not so probable, but we never know), drastic network and filesystem throughput changes, too many open connections, file descriptors, slower interaction with data systems, not enough hard disk free space for extraction, etc.

- to simplify troubleshooting and optimizations without having to request such data from the clients.

- to produce input for other components on the remote side, e.g. components within MANTA Admin GUI or currently non-existing components. As usual, most of the data have some value and our case is not an exception. In the future, Process Manager, component handling scheduling of workflows executions, could take advantage of the monitoring provided data to make scheduling more flexible, especially when we consider the multi-agent environment where a customer has multiple agents deployed. Another component could be some suggestion component, which can assist (e.g. by sorting/ranking the agents from previous runs) while the user is manually choosing the agent on which the extraction has to be executed.

1.4 Goals

The primary goals of this thesis are:

1. Get to know the Manta Flow Platform and MANTA Flow Agent

2. Analyze functional requirements for the monitoring framework for Manta Flow Agent

3. Analyze possible use cases of running multiple agents in the environment of a single customer, focusing on the management of agents and their interaction.

4. Propose a solution that fits into the Manta Flow Platform and reflects also important non-functional requirements (performance overhead, security risks).

5. Create a proof-of-concept implementation just for the scenario where the customer uses one MANTA Flow agent

1.5 Thesis Outline

In the next chapter, Chapter 2, we list collected functional and non-functional requirements, discuss about possible use cases in more detail. Chapter 3 mainly analyzes technologies and approaches that we thought that they are suitable for implementing monitoring support for MANTA Flow Agent. There is also a discussion about the final technologies selection. In chapter 4, we describe the

design of the suggested solution. Technical implementation details of the final implementation can be found in Chapter 5. There, we also describe problems that we encountered and how we resolved them. In chapter 6, we evaluate the final implementation, list known limits, and provide instructions for the usage. In the last Chapter 7, we do conclusions about our work.

2. Requirements

After the consultations with the architects of MANTA Flow Agent and Manta Platform we have collected functional and non-functional requirements which our proposed monitoring support solution for MANTA Flow Agent should fulfill. We also propose and summarize some possible use cases for introducing such monitoring support.

2.1 Possible Use Cases

There are some potential targets to monitor. Our main targets are resources used by MANTA Flow Agent and its spawned extraction processes in the multi-agent environment. We are interested in the CPU load, RAM load, hard disk usage and network related resources. We would like to also monitor developer defined specific metrics from extraction processes, have an option to easily define custom metrics at the source code level.

Other aspects could be also monitored. However, they are handled by their implementation logic and also within their error states handling. They include the basic activity status, unexpected runtime states, communication layer statuses, automatic MANTA Flow Agent upgrade progress, etc. Monitoring for MANTA Admin GUI is also out of the scope of this thesis.

Metrics could be used as an input to MANTA Process Manager, a component running within MANTA Admin GUI, which plans the executions of extractions. It could take into account the previous runs of extractions and adjust the scheduling accordingly, e.g. run multiple extractions in parallel on the same host to use effectively its resources or run extractions on different hosts, based on such information.

Another use case is to bring some overview to the user in some visual form. That from could be a simple table with basic information or a complex dashboard with graphs visualizations. The chosen form could show the current load of the system, developer-defined metrics, and historical data.

Provided output from the monitoring should simplify the troubleshooting process and identifying hot spots. By using monitoring support, we also could be able to better know the environments in which MANTA Flow Agent is operating and how effectively it utilizes available resources.

2.2 Functional Requirements

Here we list collected important functional requirements (features and functions) that the monitoring support for MANTA Flow Agent has to satisfy.

Metrics gathering

Metrics are measures of quantitative assessment commonly used for assessing, comparing and tracking some states in a system. They capture important information at a specific point in time about a process under the question.

The proposed monitoring solution for MANTA Flow Agent has to collect the following metrics:

1. JVM metrics

   JVM metrics are metrics collected from Java Virtual Machine (JVM) during application execution. MANTA Flow Agent process and its spawn extraction processes are such processes that execute within JVM.

   We are interested in JVM metrics:

   (a) Heap and non-heap memory usage of JVM. If the memory available in the JVM is insufficient, application can slow down or fail, so it's important track the usage of heap and non heap memory usage of JVM. With such metrics it is also much easier to identify and fix memory leaks.

   (b) Metrics related to JVM Garbage Collector (e.g. garbage collections count, time consumed by garbage collections). Sometimes, garbage collection is taking too long. Tracking garbage collection activity can help us to identify times when garbage collection is taking too long and also help us with garbage collection settings fine-tuning.

   (c) Number of running non-daemon and daemon threads. Those thread-related metrics could be used to identify thread leaks in the JVM, situations when application does not release references to a thread object properly.

2. Current process metrics

   These metrics are for the MANTA Flow Agent process and its spawn extraction processes and can be usually obtained from the operating system. They provide us a basic view of the currently running process from the operating system's perspective, how such process is consuming resources relative to the whole system. The following metrics we want to track:

   (a) RAM usage of the current process (e.g virtual memory and resident memory usage). This can be used to track how the application is memory-greedy over time.

   (b) CPU usage of the current process. A metric, which says how much time the application actually used from the CPU's capacity during a period, often expressed as a percentage.

   (c) Execution time spent in user space, kernel space and total elapsed time of the current process. This can help us to see how much is the application I/O bound, how big portion of time is used for I/O subsystems operations like data transferring between primary memory and various I/O peripherals (e.g. hard disk, network interface card, etc.).

   (d) Filesystem activity (hard disk throughput, number of read and write operations) of the current process. Filesystem activity shows us how often the application does filesystem manipulations such as reading and writing files and how such operations perform in time.

(e) Network activity (network throughput, number of data sends and data receives, open connections) of the current process. Tracking this activity helps us to see how much data have been transported using network interfaces, what were the maximum and average values of network throughput, how many endpoints have been connected by application, if the application correctly releases opened sockets, etc.

3. System metrics

   System metrics are metrics from the system where MANTA Flow Agent is deployed and running. They track the load of the whole system. Knowing historical overall system health has an important role when inspecting the application failures, not necessarily fatal ones.

   (a) RAM usage of the system. When the system faces memory pressure then the swapping technique can be used. In the worst-case scenario, on some platforms when the system is critically low on memory, the system's out-of-memory killer process could be employed. With that metric, we can detect if the system is close to such states, how much memory is used by the other processes and how much memory could be used for MANTA Flow Agent and its extractions processes.

   (b) Average CPU load of the system. It is the average number of processes being or waiting executed over some time period. Knowing the number of logical cores is important to interpreting CPU load averages. High load average could imply that CPU is overloaded with too many processes.

   (c) CPU usage of the system. This reflects how busy CPU is during some time interval, typically expressed by percentage. Usually, we would like to have the host CPU very close to 100%, especially if we rent instance where host is running.

   (d) Number of available logical CPUs in the system. Extraction configurations often have an option to specify how many CPUs could be used to introduce parallelism. This metric could be used in such configurations or as an dynamic input for MANTA Process Manager, the execution scheduler.

   (e) Used and available hard disk space. Extraction processes could be configured to save extracted metadata on filesystem. Those values have an informative nature for the user.

   (f) Uptime and basic operating system information (e.g. operating system family). We would like to know how long the system is running and on which platform MANTA Flow Agent is operating.

4. User-defined metrics

   In the previous points, we mentioned mainly metrics related to the system, own process and runtime resources consummation metrics. Monitoring support for MANTA Flow Agent has to bring an option for developers to define custom metrics. The developer could expose user-defined metrics via some

programming interface in the code of MANTA Flow Agent or in the code which is responsible for extraction (in extraction process). Such an interface has to be general enough to be able to model user-defined metrics, ideally not so much intrusive. E.g., such metrics in our case could be metrics related with extraction progress, extraction details (which types of entities and how many entities were extracted, how many queries to data source has been done and other internal statistics), etc.

Metrics inspection

Monitoring support has to ingest and collect metrics data. It has to be possible to inspect such metrics data. This could help us with troubleshooting, identifying the time points when some critical or suspicious events occurred. We could review metrics trends and see the state of the whole system, including MANTA Flow Agent and its extraction processes, in a big picture. The proposed metrics inspection mechanism has to fulfill the following properties:

(a) It is possible to see metrics data in real-time. The user should see the updated metrics values or new metric data when they are generated. Such data has to be deliver to the user within 30 seconds.

(b) Sampling rate for metrics data has to be within the interval 1 to 30 seconds. This value should be customizable.

(c) To inspect metrics data, a table or a graph form has to be chosen as a visualization form. One of its dimensions has to be time.

(d) It has to be possible to inspect historical metrics data. At least, the user should be able to inspect the metrics data from the previous MANTA Flow Agent and its extractions processes run. Consider introducing a customizable data retention policy for saved metrics data.

Anomalies detection

From the monitoring support, we also expect the anomalies detection mechanism. Anomalies occur on metrics data that are different from the expected ones. They could be compared to the historical values (statistically processed) or to some baseline. The support has to provide at least an option to manually identify such situations, abnormal events, but anomalies detection should be ideally triggered automatically. The following types of problems detection have to be covered:

1. Network related problems detection

   Our target monitoring applications, MANTA Flow Agent and its extraction processes, heavily use the network capabilities of the system. Without access to the network, it won't be possible to use messaging between MANTA Admin GUI and MANTA Flow Agent or its extraction processes. Also, a scanning technology very often requires access to the network for its metadata extraction. So, network reliability has an important role for us. Anomalies detection mechanism has to detect the following situations:

(a) Network throughput between MANTA Flow Agent or the extraction process and MANTA Admin GUI has significantly changed over time.

(b) Network throughput between the extractor process and its data source systems has significantly changed.

2. Filesystem related problems detection

   It's pretty common that after some time, hard disk could become the bottleneck of the system. This could be caused by some critical states like reaching the end-of-life of the hard disk, corrupted filesystem, or only by the storage characteristics. We want to detect at least the following states:

   (a) Free hard disk space on the system where MANTA Flow Agent is installed reached the user set limit. This information is meaningful as extraction processes could be configured to also save extracted metadata on the hard disk.

   (b) Filesystem read and write operations last too long compared to previous runs. This could indicate filesystem or hard drive problems.

3. Abnormal load detection

   Sometimes, it may happen that the system is overloaded. This may be caused when many processes want to finish their executions or many of them are very RAM-consuming processes. The high RAM usage can lead to swapping and in critical situations even the out-of-memory killer process can kill some processes to lower the pressure. This detection could prevent us from such situations or at least we could get the context if some fatal error occurred. Also, it could happen that MANTA Flow Agent and its extraction processes resources consumption have dramatically changed from the previous runs. Those situations have to be detected:

   (a) The high CPU load of the target system, where MANTA Flow Agent is deployed, has to be detected.

   (b) A higher usage than usual of resources (CPU usage, RAM load) of MANTA Flow Agent and its extraction processes has to be detected.

4. Potential memory leaks detection

   MANTA Flow Agent and its extraction processes run within JVM. We can consider that MANTA Flow Agent process is a long-running job. Some extractions could be considered also as long-running jobs, depending on the volume of extracted data and the complexity of the extraction process. If the application is running for a long time, memory leaks could occur and have a bad impact on the system. A memory leak is a situation when there is an object present on the heap and it is no longer used, but the garbage collector is unable to remove it from memory. Monitoring support has to actively detect such situations and warn the user about potential memory leaks caused by MANTA Flow Agent or its extractor processes run within JVM.

Programming interface for custom metrics

Our solution also has to provide a way to define custom metrics from the code and periodically export them as it is described in Section 2.2, point 4 (User-defined metrics). We have to design and implement a programming interface to expose such metrics. We should consider the usage of existing industry-stabilized APIs to model such metrics.

2.3 Non-Functional Requirements

The initial analysis also yields some non-functional requirements (system's operation capabilities and constraints). From the monitoring support for MANTA Flow Agent, it is required to fulfill the following collected non-functional requirements and recommendations:

1. Multi-platform support
   (a) It has to be possible to natively run components of monitoring support for MANTA Flow Agent on Linux and Windows operating system families.
2. Network access
   (a) MANTA Flow Agent and its extraction processes cannot be connected.
   (b) For sending metrics data, messaging should be preferred. Currently, messaging is used for communication between MANTA Flow Agent and MANTA Admin GUI, and also between extraction processes and MANTA Admin GUI.
3. Data storage, data integrity and metrics format
   (a) Data storage for metrics data has to be easily replaceable on the code and configuration level. Consider introducing some internal format, e.g. dedicated format for transporting and manipulation of metrics data, to minimize coupling to specific technology.
   (b) The monitoring support should consider specialized well supported existing solutions for storing metrics instead of creating custom solutions for data storing.
   (c) Metrics values have to be well defined and cannot be easily spoofed.
   (d) The source of metrics data has to be identifiable (multiple MANTA Flow Agents and extraction processes scenarios).
4. Usability and performance impact
   (a) The user should be able to easily configure and turn on/off the monitoring support for MANTA Flow Agent. Ideally, configuration should be done within MANTA Admin GUI, avoiding doing configuration via files.

(b) MANTA Flow Agent and extraction processes are not aware if monitoring support is enabled or disabled. When monitoring support crash or it cannot be initialized properly, then such processes should continue without problems.

(c) We should be able to extend provided solution to support monitoring for multiple MANTA Flow Agents. In this initial phase, we expect that only less than 100 MANTA Flow Agents are connecting to one MANTA Admin GUI instance.

(d) When monitoring is enabled, no significant slowdown and increased resources usage is observable.

3. Analysis

This chapter introduces existing general monitoring toolkits, their important concepts and high-level architectural overviews. Java specific technologies and frameworks which could be used to implement a custom monitoring framework for MANTA Flow Agent instances can be also found here.

Most of those technologies and toolkits are pretty standard in the field of application monitoring. We have selected the relevant ones for our use case. We took in mind also their usability and popularity (their usage in big companies, active development on GitHub, number of GitHub stars, etc.). Many of them do their own comparison to alternative solutions and this led us to discover and explore other related existing solutions, libraries.

At the end of this chapter, we discuss the cons and pros of available solutions and frameworks. We describe what they have in common, how they principally differ, and what we would have to do on the integration level, if we chose them. There, we also do a selection of technologies in which the final monitoring support for MANTA Flow Agent is implemented and explain why we chose that stack of technologies.

3.1 Monitoring Approaches and Available Technologies

In this section, we describe technologies (libraries, frameworks, programming concepts) which could be considered when implementing custom monitoring support for Java applications.

3.1.1 The Java Management Extensions (JMX)

The Java Management Extensions (JMX[1]) framework is a standard part of the Java Standard Edition platform. It provides a simple, standard way of managing resources such as services, devices (e.g. printers), and applications. JMX is dynamic, as it can be used to monitor and manage resources when they are created, installed and implemented. JMX technology can be also used to monitor and manage Java Virtual Machine (JVM).

The JMX specification [1] describes its architecture, APIs, design patterns, and services in Java language for monitoring and management of applications and networks. The overview of the architecture is shown in Figure 3.1.

To use JMX technology, one has to instrument a given resource by one or more Java objects, known as *Managed Beans* or *MBeans*. MBeans have to be registered in a managed object server (*MBean Server*).

The specification also defines the *JMX Agents* that we can use to manage any resources that have been correctly configured for management. This agent consists of an MBean Server, in which MBeans are registered, and a set of services

[1]JMX Technology Home Page: `https://www.oracle.com/java/technologies/javase/javamanagement.html`

for handling the MBeans. JMX Agents then directly control resources and make them available to remote management applications.

Other important entity is *JMX Protocol Connector*. It enables us to access JMX Agents from remote management applications. JMX connectors can use different protocols, but they have to provide the same management interface. Then, a management application can manage resources transparently, regardless of the communication protocol used. A similar holds for *JMX Protocol Adaptor*, as it provides a management view of the JMX Agent through a given protocol.

A special type of MBean, which materializes Java Virtual Machine subsystems like garbage collection, JIT compilation, memory pools and multi-threading, is *MXBean*, alternatively called Platform MBean.

To build a solution with this technology, we should keep in mind the security aspects, as this technology can use remote management applications to not only observe, but also control resources.

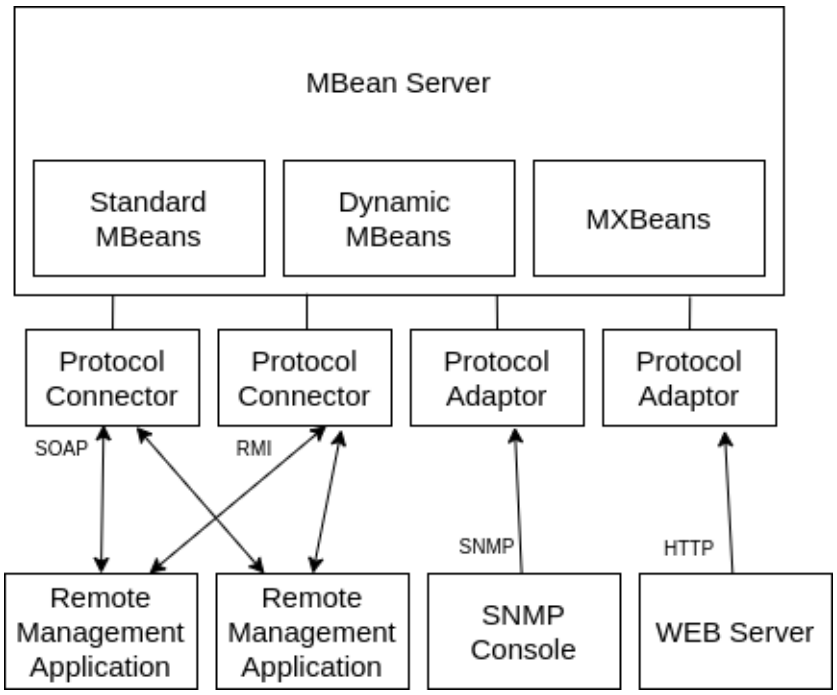

Figure 3.1: JMX Architecture

3.1.2 Spring Boot Actuator

As MANTA Flow Agent is written in Spring Boot Framework [2], an application framework and inversion of control container for the Java platform, we have also considered the Spring Boot Actuator module [2], which could help us with gathering metrics, understanding traffic and inspecting the state of the application.

Spring Boot Actuator is mainly used to expose operational information about the running application such as health, metrics, beans, configuration properties, HTTP traces, loggers, environment, etc. It uses *HTTP(S) endpoints* (REST) or *JMX beans* to expose such data.

[2]Spring Boot Framework overview is available at `https://spring.io/projects/spring-boot`

Spring Boot Actuator also provides auto-configuration for application metrics facade Micrometer (like SLF4J [3] but for metrics, see Section 3.1.9), which supports numerous monitoring systems. More about this facade and other similar API libraries can be found in Section 3.1.9.

3.1.3 Java Native Interface (JNI) and Java Native Access (JNA)

Java Native Interface (JNI) [3] is a foreign function interface (FFI) mechanism, that enables code running in a JVM to call and be called by a native application and libraries written in other languages such as C, C++.

Another way to access native shared libraries is JNA[4]. A mature library, which provides simplified access to native library methods without requiring any additional JNI or native code. Generally, JNI is faster than JNA, but JNA is much easier to use.

This can be used when some functionality cannot be implemented directly in Java programming language or just to increase application performance by using external native libraries. This include also hardware-sensitive or direct OS API calls. It can be used when we want to reuse an existing native library instead of rewriting it in Java programing language.

The huge downside of using this API is that it's against the slogan *"write once, run anywhere"*. Almost certain, we would lose the cross-platform property of Java application.

In our context, this API could be used to implement platform-specific code to collect system metrics directly from the platform (e.g via system calls). However, the mentioned disadvantage has to be taken into account.

3.1.4 Java Virtual Machine Tool Interface (JVM TI)

The JVM tool interface (JVM TI) [4] is a *native programming interface* for use by tools such as debuggers, profilers, monitoring, thread and coverage analysis tools. It provides both, a way to inspect the state and to control the execution of applications running in the Java virtual machine (JVM).

JVM TI clients are called agents. Agents can be written in any native language that supports C language calling conventions and C or C++ definitions. Agents run in the same process as JVM, and communicate directly with JVM. This communication is through a native interface (JVM TI). However, agents are often controlled by a separate process.

The agent has access to the JVM state by calling JVM TI and JNI functions, and it can register receiving of JVM TI events, using event handler functions which are called by the JVM when such an event occurs.

This API is ideal, if we would like to have a detailed overview of what the JVM is doing, to have JVM under more control. However, for our purpose, this

[3] SLF4J - Simple Logging Facade for Java serves as a simple facade or abstraction for various logging frameworks

[4] More information about JNA library is available at `https://github.com/java-native-access/jna`

is a very low-level API and its usage could cause a significant slowdown in our application.

3.1.5 Aspect-Oriented Programming (AOP)

Aspect-Oriented Programming is a programming paradigm, that can be used to add additional functionality to an existing code without modifying the code itself directly.

The unit of modularization is an *aspect*. It could be a common feature that could cover multiple methods, classes, object models, etc. Those aspects are orthogonal to each other. As aspects, we can consider e.g.: Logging, Session Tracing, Profiling, Logging, Authorization, Tracing, Metrics Collecting.

To use AOP in practice, we typically have to specify via *pointcut* specification, the locations in the original code, where code, contributing to aspect, will be inserted. This code is usually called *advice*.

In the Java world, *AspectJ*[5], *Spring AOP*[6] and *JBossAOP*[7] are widely used, and they could use different terminology, but conceptually they give a programmer similar options to use AOP approach.

Another framework which is inspired by AOP, is *DiSL*[8]. It differs from the mainstream AOP languages in the way, that user can specify any region of bytecodes to be enhanced.

In our case, this approach could be used to insert some monitoring aspects into our application. Such aspects could be e.g. tracing of the values of input and output parameters on selected methods, their execution time, and so on.

3.1.6 Bytecode Instrumentation

Instrumentation is the addition of bytecodes to methods. Usually, it is used to enhance functionality or inspect some state. Instrumentation can be used to insert code to gather data which can be then utilized by some tools. Such tools include monitoring agents, profilers, coverage analyzers, and event loggers.

In Java, AOP implementations usually use some form of instrumentation to insert aspects.

Instrumentation code can be inserted in one of the three ways:

1. Static Instrumentation. A class file is instrumented before it is loaded into the JVM.

2. Load-Time Instrumentation. The class file is instrumented when a class file is loaded by the JVM.

3. Dynamic Instrumentation. A class is already loaded into JVM (and possibly even running) is modified.

Instrumentation at bytecode level is usually done in one of this two ways:

[5]AspectJ Home Page: `https://www.eclipse.org/aspectj/`

[6]Spring AOP APIs: `https://docs.spring.io/spring-framework/docs/5.3.19/reference/html/core.html#aop-api`

[7]JBoss AOP Home Page: `https://jbossaop.jboss.org/`

[8]The DiSL Framework Home Page:`https://disl.ow2.org/`

1. using a native JVM TI native agent (see Section 3.1.4)

2. using a Java Agent (see Section 3.1.7)

In our case, bytecode instrumentation could be used within the implementation of the monitoring agent. This agent could do the similar manipulations as described in Section 3.1.5, it could add a monitoring aspect, concretely the aspect responsible for observing the application run.

Different instrumentation frameworks can provide different levels of abstraction and different levels of usability and expressiveness. The following frameworks could be used to do Java bytecode instrumentation:

BCEL

BCEL (The Byte Code Engineering Library) [5] is a low-level library that can be used to decompose, modify and recompose Java classes. It can be used in tools that do profiling, bytecode decompiling/obfuscation/refactoring, implements AOP or do a static code analysis.

ASM

ASM [6] is also as BCEL Java bytecode manipulation and analysis low-level framework. ASM can be used to modify Java classes or to dynamically generate classes. It provides some common bytecode transformations and analysis algorithms. ASM is focused more on performance (designed and implemented to be as small and as fast as possible).

BTrace

BTrace [7] can be used to dynamically trace a running Java program (similar to DTrace for OpenSolaris applications and OS). BTrace dynamically instruments the classes of the target application to inject tracing code. BTrace scripts are written in Java and they have concepts similar to AOP. Internally, it uses ASM.

Byteman

Byteman [8] is a tool that can be used to easily monitor, trace and test the behavior of Java programs and JDK runtime code. Byteman injects Java code into methods or Java runtime methods. The original program does not need to be recompiled or repackaged. This injection can be done at JVM startup or after startup, while the target application is still running. The user has to use a simple scripting language to specify when, where, and how the original Java code should be transformed. Internally, it uses ASM.

Javassist

Javassist [9] is a library for bytecode manipulation which can be used to define a new class at runtime and modify a class file when the JVM loads it. It provides two levels of API: bytecode level and source level.

The bytecode-level API allows the user to directly modify bytecode as in ASM or BCEL. The source-level API makes it possible to edit a class file without any

knowledge Java bytecode. Inserted bytecode can be in the form of Java source, as Javassist compiles it on the fly.

Byte Buddy

Byte Buddy [10] is another bytecode manipulation library, which can be used to do code generation and bytecode modification during the runtime of a Java application. To use Byte Buddy, the user doesn't have to understand the class file format or Java bytecode. It offers a convenient API for building and modifying classes, either manually or using a Java Agent. Internally, it uses ASM.

Java Native Instrumentation Framework (JNIF)

JNIF [11] is a C++ library which can be used for Java bytecode manipulation. It is similar to frameworks like ASM or BCEL, but instead of being implemented in Java, it is implemented in C++. The big advantage of JNIF is that it can be used for implementing JVM TI agents (see Section 3.1.4). So, JNIF enables the development of dynamic analysis tools that require full bytecode coverage and minimal perturbation.

3.1.7 Java Agent

Java agents [9] are part of the Java Instrumentation API (*java.lang.instrument* [12] package), and they can instrument programs running on the JVM. Java Agent is a jar file, it contains *premain* method and has special attributes specified in its manifest. The premain method is called after JVM has been initialized and must return to process the startup sequence.

Instrumentation API can be used to redefine or retransform classes during run-time. The user can change method bodies, the constant pool and attributes. However, the retransformation or redefinition must not include removing, adding or renaming fields or methods. It is also not permitted to do changes in the signatures of methods or inheritance.

Java agents together with bytecode instrumentation are commonly used to implement AOP, profiling, mutation testing, and also some form or a part of monitoring.

Application Performance Monitoring (APM)

APM refers to the management of software application performance to ensure an expected level of service (measured by metrics). In our context, we are interested in APM for Java applications. Many of them also use Java Agent. Usually, its role is instrumenting the target application, adding code to collect data, and exposing collected data. Some of them expose such data via an embedded web application or just send them to some remote part of APM, where they are processed.

Such agents often can instrument selected popular frameworks and libraries like Hibernate, Spring Framework, HTTP clients, JMS, JDBC, Log4j, Kafka, etc. If the target application uses such a framework or library, then components of

[9] A simple Java Agent implementation can be found at `https://www.baeldung.com/java-instrumentation`

such framework are instrumented to record the usage of the framework. Some agents can also observe application servers such as Tomcat, Wildfly, Jetty, etc.

There are many APMs for Java application that uses Java Agent. Currently, some commercial options, such as FusionReactor, Datadog, Sematext, Dynatrace, New Relic, Retrace from Stackify, Instana, exist. Some of them have Java Agent implementation open-sourced.

Of course, there are also fully open-sourced and not commercial solutions. E.g. Glowroot, Pinpoint, Scouter, Stagemonitor, inspectIT Ocelot.

It's also common that they use Byte Buddy for the instrumentation in their Java Agent. It's easy to learn and use and also less complicated than ASM or similar frameworks.

Those solutions usually differ in the level of configurability, frameworks and libraries which they can instrument, how their Java Agent could be extended (communication protocols, data storage technology support, own plugins support to extend their functionality), if they can be reloaded during runtime, if they supply some API to manually instrument the target application, etc.

3.1.8 Libraries for System Metrics

We are also interested in *system metrics*. System metric is a metric collected, computed from the system, where the application is installed and run. Typically, those metrics are gained from the operating system (directly or indirectly). Examples such metrics are memory usage, network activity, number of running processes, etc.

To gain those metrics, we prefer to use a Java library which could handle different platforms, as MANTA Flow Agent can run on different platforms.

One candidate for such a library is SIGAR (System Information Gatherer And Reporter)[10], which supports different operating systems. It is written in C with bindings to JAVA. Unfortunately, this project seems to be abandoned.

Another Java library, which can be used for fetching system metrics, is OSHI (Native Operating System and Hardware Information)[11] library. It is a free JNA-based (native) library for Java. No additional native libraries are required and it provides a cross-platform implementation to retrieve system information, such as OS version, memory, processes and CPU usage, etc.

Linux and Docker

We have also considered an option to get system metrics from the operating system via reading specific files on pseudo file systems. This is also a way how to dig some operational information about the current process or information about the whole system from the operating system. However, this is typically done with other libraries that also abstract system metrics, so, the user of the library doesn't have to deal with the target platform at all. We also considered getting some metrics from the MANTA Flow Agent run within a container but we didn't find anything interesting which could be observed from the inside of the container.

[10]SIGAR project can be found at: `https://github.com/hyperic/sigar`

[11]OSHI project can be found at: `https://github.com/oshi/oshi`

3.1.9 SDKs and APIs for Metrics Instantiation

Many monitoring frameworks provide a way to define metrics within the user code. The user usually has to use some client library for that purpose. Those libraries usually provide some classes/interfaces or annotation based APIs. They can be used to instrument a target application. Within this section, the word *instrumentation* has a different meaning than in bytecode instrumentation. By instrumentation, we mean taking measurements from an application.

However, there exist also a monitoring framework-agnostic metrics instantiation APIs/SDKs such as OpenTelemetry or Micrometer.

OpenTelemetry

OpenTelemetry [13] is a collection of SDKs, APIs, and tools. It can be used to instrument, collect, generate and export data such as metrics, traces and logs. It tries to be vendor-neutral, supports many languages, and also integrates with Spring framework. OpenTelemetry combines metrics, logging and distributed tracing into a single set of system components and language-specific libraries. It combines and replaces the usage of OpenTracing (APIs and instrumentation for distributed tracing) and OpenCensus (collecting metrics and distributed trace), which are both deprecated now.

Micrometer

Micrometer [14] is a vendor-neutral application metrics facade for Java programs. It is also integrated into Spring framework (see Section 3.1.2) and it supports many popular monitoring systems such as Datadog, JMX, Dynatrace, Prometheus, Graphite, Influx. It provides interfaces for counters, gauges, timers, distribution summaries, and long task timers and supports a dimensional data model.

Metrics Java library

Metrics [15] is another Java library for capturing application-level and JVM metrics. In addition to modeling metrics via gauges, counters, histograms and timers, it supports reporting via JMX, HTTP, CSV files and also Graphite database.

Client libraries

Many existing monitoring solutions also provide their own instrumentation API or SDK to easily instrument Java applications. Some of them also provide support for the previously mentioned vendor-neutral API. We should consider the usage of such client libraries if some existing monitoring solution is chosen.

3.2 General Monitoring Solutions

In the previous section, we have listed Java specific technologies which could be used to implement monitoring support for MANTA Flow Agent. In this section we provide existing general monitoring solutions and also introduce Time Series

Databases which are suitable for building a custom monitoring solution on the top of them. We also mention Online Analytical Processing solutions as an alternative data technology for metrics data, as some time series implementations do a performance comparison to some selected Online Analytical Processing solutions.

3.2.1 Pull and Push model

Monitoring solutions have to deal with metrics data. This data have to somehow get into the monitoring system. We can distinguish two major monitoring architectures:

1. *Push model* - metrics are periodically sent by each monitored component to a central collector, usually via UDP.

2. *Pull model* - a central collector periodically requests metrics from each monitored component, usually via HTTP.

Many monitoring solutions implement primarily one of those models. There is no clear winner. Both have pros and cons in specific situations.

In the Pull model, it is easier to control the authenticity and amount of data as the puller chooses the time and monitoring targets to scrape metrics data from. However, with this approach, it's not so straightforward to collect metrics from the short-lived batch jobs. In the Push model, those jobs could directly send metrics data to the collector. Although, in this model, the collector could become the target of a denial of service attack.

The Push model makes usually the discovery of new monitoring targets easier than the Pull model. In such a model, a new monitoring target has to just notify the collector about its existence, e.g. it just sends the metrics data directly to it. Such registering mechanism must be carefully analyzed and implemented, as it could create space for security accidents.

The Pull model doesn't fit very well for an event-based monitoring system. The system, where each individual event should be immediately reported to a central monitoring system. Naturally, the Push model looks like it is a better option for such systems.

We will see that both models are popular and they have their representatives in this chapter. Deeper discusions can be found in [16], [17], [18] and [19].

3.2.2 Time Series Databases

In general, metrics data are usually saved in some time series database. Time series database is a database that is optimized for storing and serving time series, timestamp associated data. More introductory text can be found in [20] and [20].

Time series databases usually support multi-dimensional data, e.g. via attached tags to the inserted data, to enable fast queries over dimensions. Such dimensions could represent e.g. a data center location of the server, the server's CIDR block, if the server is a production or development machine, etc.

To get data from a time series database some query language is implemented. We will see that such query languages are based on SQL (enhanced SQL) or they are designed from the ground to be more close to time series database's data

model. Such languages are often used not only to query data but also to define expressions that participate in alerting component, e.g. they are periodically, on-demand or when data arrived evaluated and checked if the value is between limits, and if they are not, then an alert is fired.

The way, how data are written into time series database depends on the implementation.

OpenTSDB

OpenTSDB [21] is an open-source, scalable, distributed time series database written on top of open-source HBase, Java non-relational distributed database. The core component is Time Series Daemon (TSD). Metrics are pushed into TSD, so the Push model is implemented, as we can see in Figure 3.2:

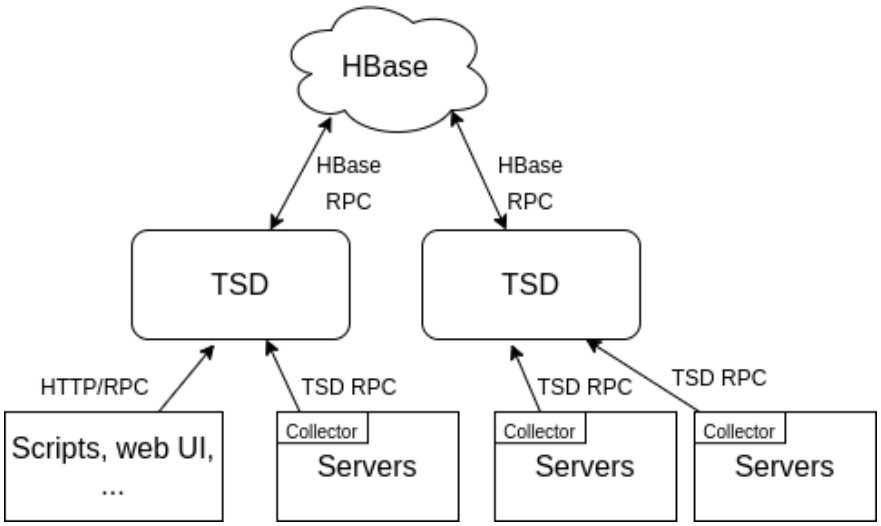

Figure 3.2: OpenTSDB Architecture

A time series data point consists of a metric name, a timestamp (epoch time), a value (integer, floating-point, JSON) and optional set of tags (key-value pairs) as it's shown in Listing 3.1

```
cpu.load_1m 1647693790 2.12 dataCenter=dc1 server=web01
cpu.load_5m 1647693790 1.52 dataCenter=dc1 server=web01
cpu.load_15m 1647693790 1.40 dataCenter=dc1 server=web01
cpu.overloaded 1647693790 1 dataCenter=dc1 server=web01
```

Listing 3.1: Metrics written in OpenTSDB format

To communicate with the TSD, a simple telnet-style protocol, a REST API or a simple built-in GUI can be used. To read data, e.g. to external systems such as monitoring frameworks, dashboards, statistics packages or automation tools, REST API is available.

QuestDB

QuestDB [22] is an open-source SQL time series database built for performance, written in Java. It is a relational column-oriented database designed for time

series and event data. For inserting data one can use InfluxDB line protocol (see Section 3.2.3), Postgres wire protocol, or Java API if it is used in embedded mode. REST API can be used for importing data and executing an SQL statement.

QuestDB can integrate with third-party tools and utilities for collecting metrics and visualizing data such as Grafana (visualization, see Section 3.3.1), Kafka (ingesting data from Kafka topics) and Telegraf (uses QuestDB as data storage).

TimescaleDB

TimescaleDB [23] is an open-source relational database for analytics and time series. It supports full SQL and introduces new capabilities for time series data management, new query planner and query execution optimizations, new functions for data analytics. TimescaleDB is built on top of PostgreSQL and is packed as a PostgreSQL extension as we can see in Figure 3.3.

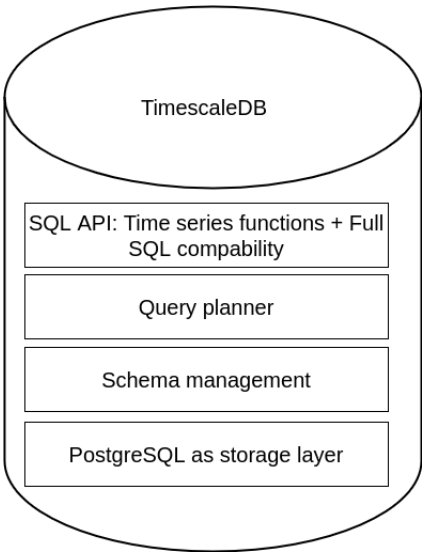

Figure 3.3: TimescaleDB

One interesting open-source project which is built on TimescaleDB is *Promscale*[12]. It is a storage system for metrics from Prometheus, the monitoring platform. We will talk about Prometheus in this chapter too. To query data from Promscale we can use PromQL or SQL from TimescaleDB.

[12]More details about Promscale project is available at: `https://github.com/timescale/promscale`

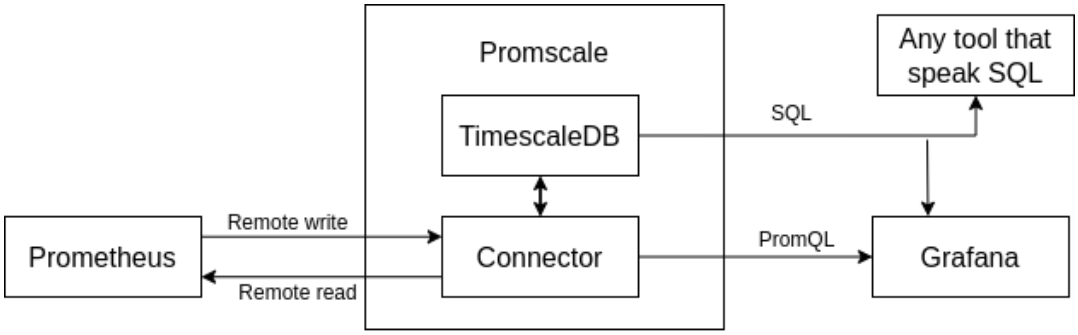

Figure 3.4: Promscale Architecture

Graphite

Graphite [24] is an open-source time series database that stores numeric time series data and renders graphs of this data on demand. It was initially released in 2008 and it is an older time series database. Graphite is written in Python 2 but native Graphite on Windows is completely unsupported. It supports only the Push model for feeding the database with metrics.

Graphite consist of 3 software components. Their roles and interactions is shown in Figure 3.5. Those components are:

1. *Carbon* - a daemon which listens for time series data (an event-driven networking engine).
2. *Whisper* - a simple database library for storing time series data (allows the insertion of multiple data points at once).
3. *Graphite* web app - Python (Django) web app renders graphs on-demand.

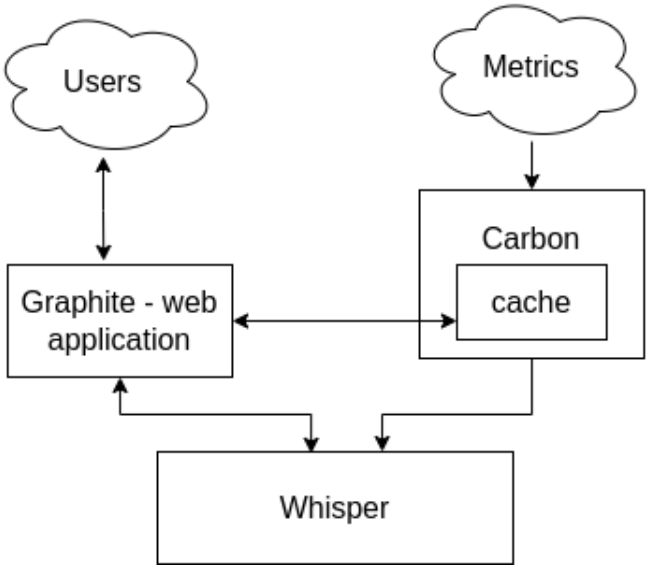

Figure 3.5: Graphite Architecture

A Graphite metric consists of a metric name, a metric (numeric) value and a metric timestamp (epoch time). Graphite also supports tags that could be appended to the metric name as we can see in the Listing 3.2:

```
collectd.host.load.longterm 2.3 1647699790
disk.used;datacenter=dc1;server=web01 80.2 1647693790
accounts.authentication.attempted 2 1647699790
```

Listing 3.2: Metrics written in Graphite format

The following three protocols could be used to encode such metrics before sending them to Carbon:

1. The plaintext protocol.
2. The pickle protocol, Python binary object serialization and de-serialization support - recommended method for sending large amounts of data.
3. Advanced Message Queuing Protocol (messaging middleware) - Carbon listens to a message bus.

The standard way of retrieving raw metrics data from Graphite is REST API. The Graphite web app could be used to visualize metrics directly, or one could use its render API to generate graphs.

There exist many tools that work with Graphite. Those tools could integrate in different levels such as metrics data collection (Graphite as data storage), forwarding metrics data into a different system, visualizing metrics.

3.2.3 Monitoring Toolkits

In this section, we present high-level overviews of the most popular general monitoring toolkits which can be used to implement a custom monitoring platform.

All of them implement their own time series database which can be used as a standalone component for storing metrics data.

Influx

Influx is very popular open-source time series platform [25], toolkit. The commercial support is available. With the Influx toolkit, we can write metrics data, query data, process data and visualize them. It implements Push model, but to imitate Pull model the Telegraf [13] component can be used. We have to distinguish versions of Influx platforms, as version 2.0 brings important architectural changes. One of them is introducing the new query language *Flux* (functional scripting and query language optimized for monitoring, alerting, ETL) and deprecating *InfluxQ*, the old SQL-like language for time series data. The older version of InfluxDB was the part of the TICK Stack[14] (visualized in Figure 3.6) and consists from those components:

- *Telegraf* - an agent which can be configured to use plugins to scrape metrics from targets or generate metrics, and send them to InfluxDB instance.

[13]More details about Telegraf is at `https://www.influxdata.com/time-series-platform/telegraf/`

[14]More information about TICK Stack (InfluxDB 1.x) can be found at `https://www.influxdata.com/time-series-platform/`

- *InfluxDB* - time series database.
- *Chronograf* - the user interface and administrative component for InfluxDB and Kapacitor.
- *Kapacitor* - a real-time streaming data processing engine. It could be integrated with any anomaly detection engine or alerting framework.

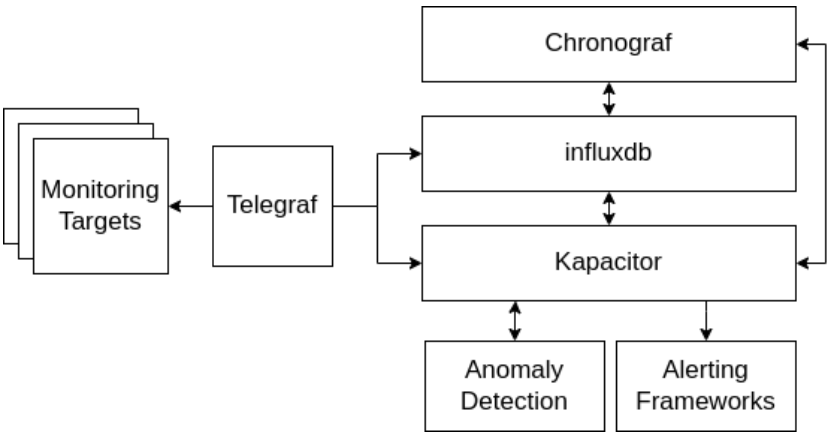

Figure 3.6: Deprecated TICK Stack with InfluxDB 1.x

From version 2.0, InfluxDB[15] has integrated the functionality of Chronograf and Kapacitor. InfluxDB daemon also serves web application where the user can use such functionalities.

Data could be loaded into InfluxDB via UI of InfluxDB web app, via clients libraries (many languages already supported, including Java) or by Telegraf, which pushes data into InfluxDB.

InfluxDB uses *line protocol* to write data points. A InfluxDB metric consists of a metric name, an optional tag set (key-value pairs), a field set (key-value pairs, at least one pair) and an optional metric timestamp (epoch time). Values in fields set must have boolean, integer, unsigned integer, float or string type. Data written in this format are shown in Listing 3.3:

```
cpuLoad,dataCenter=dc1,server=web01 load1m=2.12,load5m=1.52,
    load15m=1.40,overloaded=true 1647693790
memUsage,dataCenter=dc1,server=web01 total=16106127360,used
    =2576980377 1647693790
```

Listing 3.3: Metrics written in InfluxDB format

InfluxDB client libraries are recommended to use to query data (using the Flux language) and write data (line protocol or POJO). Those client libraries just call REST API on InfluxDB.

[15]The current architecture of Influx platform can be found at `https://www.influxdata.com/solutions/application-performance-monitoring-apm/`

Prometheus

Prometheus [26] is another open-source monitoring toolkit, time series database with a multi-dimensional data model, own query language (*PromQL*) and alerting support.

To get metrics into Prometheus, Prometheus has to be configured to scrape metrics from the targets (jobs, exporters), so, the Pull model for data feeding is implemented. Targets expose all their metrics via one HTTP endpoint. However, it's possible to use the optional Prometheus component *Pushgateway* to allow ephemeral and batch jobs to expose their metrics by pushing them into Pushgateway. Prometheus is then configured to pull, scrape, such data from the Pushgateway. Prometheus also implements alerting via the optional separate process *Alertmanager*. To visualize collected metrics built-in web UI in Prometheus can be used, but it's more common to use other external visualization tools such as Grafana (see Section 3.3.1).

In the next Figure 3.7 we can see the basic components of the Prometheus toolkit, integrations to external systems and their interactions:

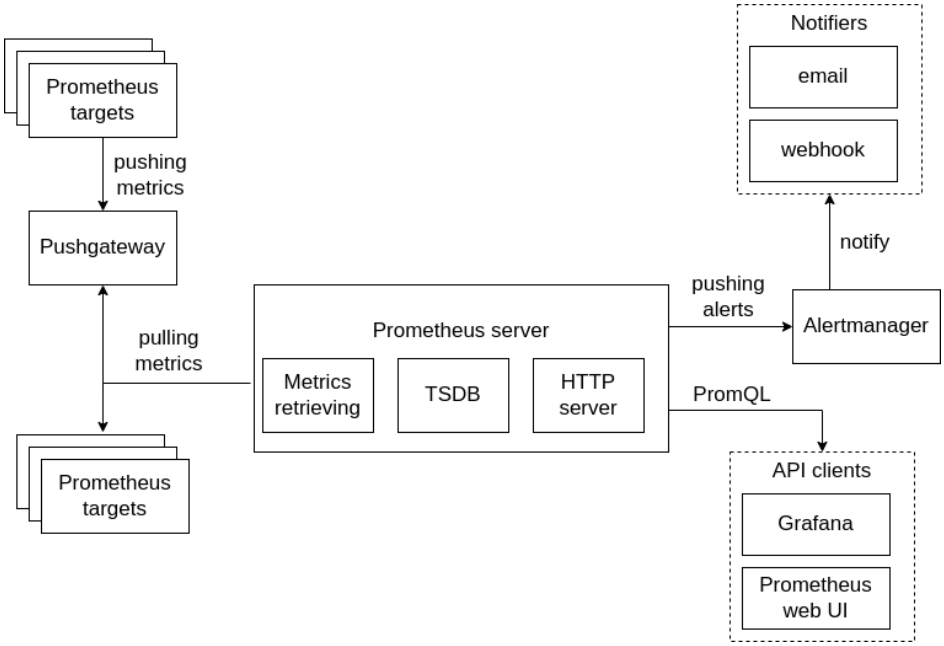

Figure 3.7: Prometheus Architecture

A Prometheus metric consists of a metric name, an optional set of labels, a metric float value and a metric timestamp (epoch time) as it is shown in Listing 3.4:

```
cpu_load_1m{dataCenter="dc1" server="web01"} 2.12 1647693790
cpu_load_5m{dataCenter="dc1" server="web01"} 1.52 1647693790
cpu_load_15m{dataCenter="dc1" server="web01"} 1.40 1647693790
cpu_overloaded{dataCenter="dc1" server="web01"} 1 1647693790
```

Listing 3.4: Metrics written in Prometheus format

Prometheus provides also client libraries for many programming languages

which can be used for application instrumentation to collect metrics from the application and expose them through the HTTP endpoint.

3.2.4 Online Analytical Processing (OLAP) solutions

With Online Analytical Processing tools [27] we can analyze multidimensional data interactively from multiple perspectives. As we saw in previous subsections, metrics data often models multidimensional data via some form of tags, labels. We have also explored the following OLAP databases as a potential option for storing and querying such metrics data.

ClickHouse

ClickHouse[16] is an open-source column-oriented database management system that allows generating of analytical data reports in real-time. It tries to be the best in the industry query performance, while significantly reducing storage requirements through. It provides SQL interface for data operation. ClickHouse is horizontally scalable and implements fault tolerance.

Apache Pinot

Apache Pinot[17] is an open-source real-time distributed OLAP datastore designed for the execution of real-time OLAP queries with low latency on massive amounts of data and events. It is column-oriented database which supports SQL for querying.

Apache Druid

Apache Druid[18], similarly as Apache Pinot, is an open-source high-performance real-time analytics database with SQL support intended for enterprise deployment. It is more aware of high availability and multitenancy support.

3.3 Post-Processing and Integrating Metrics

As we saw in the previous section, many existing monitoring solutions not only collect metrics but also integrate with other independent components, systems. Such components could be external or embedded. They could provide:

- visualization of data. Provided monitoring solutions often support at least an embedded web application in which it's possible to visualize current or historical data. Some of them also provide endpoints to directly generate graph images or a public API to retrieve data for other visualization frameworks, toolkits.

[16] An introduction to ClickHouse is available at: `https://clickhouse.com/docs/en/intro`

[17] Apache Pinot in more details is described at `https://docs.pinot.apache.org/`

[18] When to use Apache Druid is described at `https://druid.apache.org/docs/latest/design/#when-to-use-druid`

- anomalies detection and alerting. Anomalies detection identifies when a metric is behaving differently than it has in the past. Alerting is the responsive component that performs actions based on changes in metric values. It should be flexible enough to notify operators. Mentioned monitoring tools often offer such functionality either via configuration interface or via programing interface.

3.3.1 Visualization tools

To visualize metrics data some external tools could be used. They support the wide range of different data sources, or one could implement some adapter if the direct support for a data format is missing.

Grafana

Grafana [28] is one of these softwares. It is an open-source web application that is mainly used to easily visualize and query data. Grafana supports many different storage backends for time series data - data sources. OpenTSDB, TimescaleDB, Graphite, InfluxDB and Prometheus are also officially supported. Each data source can have a specific query language and that language can be used in Grafana to query data for visualization in Grafana dashboards. Grafana is very extensible. The user can provide their plugins to support new data sources and they can define a new type of visualization panel.

With Grafana, it's also possible to define *alerting rules* [19]. Rules define evaluation criteria that determine whether an alert will fire and it consists of one or more queries. When an alert is triggered, the user could be notified via different notifiers, e.g by an email, Discord, Kafka, Pagerduty, Slack, or webhook.

In the next Figure 3.8 we can see some dashboard created in Grafana to visualize system metrics.

Figure 3.8: Grafana dashboard

[19] Alerting options of Grafana can be found at: `https://grafana.com/docs/grafana/v8.4/alerting/`

3.3.2 Anomalies Detection and Alerting

Modern motoring tools are able not only to collect data (e.g. for real-time data visualization and historical inspection) but they also provide some kind of anomalies detection and alerting possibilities. Solutions in Section 3.2.3 support only static anomalies detection, where the user has to specify what is the critical value. Dynamic anomalies detection solutions, where the monitoring system can automatically determine the critical values, e.g. via machine learning, weren't analyzed.

The built-in support or an external optional dependency of such solutions could be separately used to bring such features. Namely *Alertmanager*[20] from Prometheus toolkit (Section 3.2.3), Kapacitor from Influx platform (Section 3.2.3) or we could use such functionality from visualization component (e.g. Grafana, Section 3.3.1).

3.4 Discussion and Technologies Selection

In this section, we provide technologies selection which are finally used in our monitoring support for MANTA Flow Agent. We compare how suitable they are to implement such support, their cons and pros, and which difficulties we would have to overcome if they were chosen. After analyzing the technologies, toolkits and libraries mentioned in previous sections, we came to the conclusions summarized in this section.

3.4.1 JMX and Spring Boot Actuator Usage

We cannot use JMX directly as-is. As we have stated in non-functional requirements (Section 2.3), we cannot connect to MANTA Flow Agent or its extraction processes from the remote side. JMX also brings remote management capabilities and this could be prone to security accidents. Such remote management capabilities enable remote applications to access the MBean Server through JMX Connectors and JMX Adaptors. To use the full capabilities of JMX in our scenario, we would have to implement a custom JMX Adaptor for MBean Server on MANTA Flow Agent side to tunnel JMX communication to the remote side via messaging. Messaging is the preferable way to implement such communication, see non-functional requirements (Section 2.3). On the remote side, there would be JMX Connector which could provide the standard interface for JMX remote management and transparently tunnel queries to the adaptor. This solution seems to be over complicated and it is against the clear and recommended architecture of JMX. It could bring problems when considering multiple MANTA Flow Agents such as aggregation of multiple instances within JMX Connector on the remote side. So, finally, JMX was not chosen as a technology for managing and monitoring our applications. However, in the final solution we use MXBeans, a special type of MBean, from the JMX ecosystem to gather metrics from JVM, but without any JMX Connector or JMX Adapter use. With MXBeans, we can easily access JVM metrics mentioned in the functional requirements (Section 2.2).

[20] Alertmanager integration with Prometheus is described at: `https://prometheus.io/docs/alerting/0.23/alertmanager/`

Another interesting solution could bring us simple out-of-the-box monitoring support is Spring Boot Actuator module from Spring framework. However, it supports only HTTP(S) endpoints and JMX beans to expose metrics data. But, as it was said in the previous paragraph, we do not want to use JMX and also make it possible for MANTA Flow Agent to be connected.

3.4.2 APM Solutions

In this chapter, we also mentioned some Application Performance Monitoring (APM) solutions specific for Java programs. We found also some open-source implementations. Most of them cover our basic expectations for monitoring support and offer many additional features (online configuration reloading, different storage technologies support, API for custom metrics, instrumentation for popular frameworks and application servers, etc.). They usually use Java Agent (see Section 3.1.7) at least to gather metrics. Some of them can send metrics data to an external system (e.g. paid metrics collection platforms). Other ones don't send such data, they only expose them to the user via an embedded web application. In addition to visualization of data, within this web application, it is often also possible to configure anomalies detection and alerting.

However, we didn't find any suitable APM solution which could be simply integrated and fulfill our important requirements related to the communication constraints. Many of them are not flexible at this critical point and do not support an option to modify metrics data flow easily, e.g. via custom commutation adaptors.

In the first case, when Java Agent does not send or communicate with external systems, we could reuse them only for metrics gathering and disable embedded web application. In this way, we would lose all functionality that its embedded web application brings and we would have to implement some transport adapter similarly to the JMX case, described in previous paragraphs. Then, those functionalities would be handled by another existing component of the existing solution or it would be coded from the ground.

The same approach could be used for the second case when Java Agent sends or communicates with an external system. In this case, Java Agent is usually lightweight and doesn't serve any web application, only gathers metrics and sends them to some external system.

In both cases, it wouldn't so straightforward to overcome such obstacles. We also would have to be aware of licensing and maintainability of such modified solutions. So, the reuse of a Java Agent implementation of an existing APM was not acceptable.

3.4.3 Time Series Databases and OLAP Solutions

We considered also not Java specific monitoring solutions as well. We found out, they internally use a specialized database, Time Series Database, optimized for storing and serving metrics data (time series).

One of them is Graphite. This solution wasn't chosen as it is not officially supported on Windows platform. Otherwise, we could use its component Whisper, a simple database library for storing time series data, and probably the Graphite

web application could be also integrated. To get data into the Graphite database we could directly use Whisper or run the third, the last component from the Graphite stack, Carbon, a daemon that listens for metrics data. In the next paragraph, we will see how such metrics data could be gathered. Then, metrics data would be pushed into Carbon from MANTA Admin GUI. We would have to manage to get gathered data from MANTA Flow Agent and its extraction processes to MANTA Admin GUI. Graphite doesn't implement any anomalies detection or alerting functionality. It only stores numeric time-series data and renders this data on demand. So, other functionalities would probably be supplied by integrations with external components. For example, Grafana supports Graphite as a data source and Grafana can do visualization and alert management as well.

Next, we have OpenTSDB, distributed time series database. The situation is similar to Graphite, as it only stores and serves massive amounts of metrics. It consists of two components: Time Series Daemon (TSD), which can be instantiated multiple times, and Hbase (non-relational distributed database). We could run TSD on MANTA Admin GUI side and gather and tunnel metrics data from our applications (MANTA Flow Agent and its extraction processes) similarly to the Graphite case, and then push those metrics data from MANTA Admin GUI to TSD. Visualization and alert management could be done within Grafana, as Grafana supports OpenTSDB, or could be implemented by hand. We didn't choose OpenTSDB, as it is targeting a higher volume of data and to be run in a distributed environment. Using distributed HBase is complex and we wouldn't benefit from it, as we do expect a higher volume of data.

A more advanced monitoring toolkit is the Influx platform. With this platform, we have multiple options. We could use the whole stack, Telegraf (collecting agent) and InfluxDB with its embedded web application. Within this application, we can visualize metrics data and also use the functionality of embedded real-time streaming data processing engine (in the older architecture known as Kapacitor) to implement custom anomalies detection and to integrate with the alerting frameworks (all by using internally its query language Flux). However, Telegraf would have to be placed on MANTA Admin GUI side and we would have to configure Telegraf to scrape metrics data from MANTA Admin GUI. This data could get there the same way we described in the previous paragraphs. A better option is skipping the presence of Telegraf at all. Instead of that, metrics data could be filled into InfluxDB directly from MANTA Admin GUI. There is an official Java client library for that purpose. Alternatively, for visualization, anomalies detection and alerting, InfluxDB could be integrated with another external system (e.g. Grafana supports InfluxDB). This is the first serious candidate which we should consider when building our monitoring support.

In the previous paragraphs, we saw that we usually cannot use the suggested monitoring frameworks as-is and we would have to somehow customize them to use in our environment. There is also an option to use only their time series database in standalone mode and integrate them with other custom or existing solutions. To build a custom lightweight solution we could use QuestDB, an SQL time series database built for performance (written in Java, so it could be used in embedded mode easily) or TimescaleDB time series database. Both of them use SQL-based query language and they do not provide any additional functionalities.

They are only time series databases. However, they support many integrations, at least they have Java client libraries. One interesting integration is that QuestDB could be used as a drop-in replacement for InfluxDB. So, in the previous paragraph, we could consider also a scenario where we could use QuestDB which could communicate with other external solutions via InfluxDB APIs. Another interesting integration option provides Promscale, which is built on TimescaleDB. With this, we could in real-time import metrics from Prometheus and expose them to other systems. Those systems can use SQL and PromQL (query language from Prometheus) to quey this data. Basically, it could be used as a scalable long-term storage system for Prometheus metrics and make them observable via SQL. Conclusively, we don't see the need to use any of those time-series databases or their adapters, especially as a starting point, over more standard and well-supported time-series databases.

As an alternative to a time-series database, we could potentially use some Online Analytical Processing (OLAP) solution. We find out, that this is not so good idea and it does not provide benefits over the standard time-series database. They are well-prepared for a huge amount of data, distributed deployment, high dimensions and high-performance. They also do not well integrate with other solutions and require much more resources for their run as a standard time-series database. So, we excluded this as an option.

Another widely used monitoring toolkit is Prometheus. Its core component is Prometheus server. Optional ones are Alertmanager and Pushgateway. As Prometheus uses the Pull model, feeding metrics data is not so straightforward in our scenario. We could use the standalone component Pushgateway for sending metrics data from MANTA Admin GUI. Such metrics data could be retransferred from MANTA Flow Agent and its extraction processes to MANTA Admin GUI. Alternatively, we could implement our own HTTP(S) endpoint to expose such data on MANTA Admin GUI side. Then Prometheus could be configured to scrape data from it. To provide anomalies detection and alert management, Alertmanager or Grafana could be used.

Finally, we were deciding between Influx platform and Prometheus. They differ mainly in the primary supported metrics data feeding model (the Pull model in Prometheus vs the Push model in Influx platform). Both of them are well supported and provide a good ecosystem (integration with other systems and tools, client libraries, clean stable API, composable architecture, etc.). InfluxDB metrics data type spectrum is wider and better suitable for event logging, while Prometheus metrics data has to have float value. Both support multidimensional data. Prometheus via labels and InfluxDB via equivalent tag set. Both are pretty standard in the world of application monitoring. After this discussion, we finally chose Prometheus as time-series database for our monitoring support. In the future, MANTA Platform would be more micro-services-oriented and alerting support could be easier achieved with the Alertmanager from the Prometheus stack. While trying to implement such functionality in Influx would require more time and it wouldn't be so straightforward. How Prometheus is finally integrated into our monitoring support is described in the next chapter.

3.4.4 Monitoring API and Metric Gathering

One of our functional requirements is that we have to provide an API for the user-defined metrics. This could be achieved in multiple ways. We could use for that purpose Java client library for Prometheus. However, this client library is designed to work in a way that it creates HTTP(s) server and expose such metrics and this is not desired. There is an option to avoid this, but then the usage of such API wouldn't be so pleasant. It's not possible to change the default behavior.

Alternatively, we could use some other standardized API for custom metrics instantiation. Micrometer is an example of such API. It looks promising, as it is a metrics facade for Java applications. We prefer not to be so much coupled with concrete data storage technology, in our case with Prometheus, and API should be vendor-neutral. In the end, we conclude that we should design our own API as most analyzed APIs are pretty simple, they don't bring many benefits and they could be easily imitated.

So, instead of the existing client library for Prometheus metrics or vendor-neutral API, we decided to design our own API. We will see more details in the next chapter. This gives us more freedom as we can design API more specialized for our scenarios and context.

To implement a custom API, we decided to give bytecode instrumentation a try. Alternatively, we could use some complex AOP framework for that purpose. E.g. by introducing some API, like artificial classes and annotations, which could be a part of pointcut specifications. Code doing some observations and registering monitored entities would be in an aspect role. For us, it was easier to insert such code manually, via a more low-level approach, via bytecode instrumentation.

To insert such code, we preferred Dynamic Instrumentation, but finally, we ended with Load-Time Instrumentation. Many frameworks could be used for that purpose. We exclude BCEL and ASM, as they are very low-level. In this chapter, we also mentioned two instrumentation tools, BTrace and Byteman. Both of them are not suitable for implementing our custom API, as they use some scripts written in their scripting language or Java and those are not so flexible in the sense that they are scripts. They couldn't be easily used to bridge or define our custom API. Then, there are other two suitable instrumentation libraries, Javassist and Byte Buddy. Both of them highlight their useability. To use Javassist more simply, one has to write Java code into Java String to do instrumentation, which is not so type-friendly. In Byte Buddy, that all can be written directly in Java. We ended with the Byte Buddy, as it is also the most used framework used in Java Agent for APMs, and those agents also do similar things we expect from our API.

To collect metrics from JVM we decided to use MXBeans from JMX ecosystem. It seems to us that the provided level of observability of running JVM exposed by JMX is sufficient to fulfill our requirements. If we wanted to have a deeper insight into what the JVM is doing, the JVM TI could be used. However, the interface is so low-level and it could significantly slow down the application if it is used incorrectly.

For system metrics gathering, the JNI or JNA interfaces could be used to implement platform-specific system metrics collection. To avoid this, we prefer to use an existing library which does it for us. We selected the OSHI library (see

Section 3.1.8), which is capable of capturing the most important system metrics and process metrics (see Section 2.2).

In general, to gather metrics from our applications, MANTA Flow Agent and its extraction processes, we decided to use a custom monitoring Java Agent. It would be responsible for collecting all types of metrics (JVM metrics, system and process metrics, user-defined via our API) and sending them to MANTA Admin GUI. That communication would be done with messaging.

3.4.5 Visualization, Anomalies Detection and Alerting

As we use Prometheus as a time series database, there are already some integrations that could handle this for us.

To visualize metrics data to the user we could use one of the following four options. As Prometheus also serves web application which is capable of doing that, this is also an option. We do not recommend this option, as its visualization capabilities are very limited and they should be used mainly for debugging. Next, we could use Grafana as a visualization application. There, we considered two modes. In the first case, Grafana can run as a service that is available to the user. In the second mode, where Grafana is not available to the user directly and it is used only as a graph engine that renders graphs. Such graphs could be embedded in our custom front end. Lastly, we could implement a custom front end for visualization, but to visualize metrics data some other library that can draw graphs from Prometheus data could be used.

Anomalies Detection and Alerting management could be supplied by Alertmanager, Prometheus optional standalone component. With Prometheus and this application, we would be able to configure alerting rules and define alerts consumers. Another way how we could achieve similar behavior is to use Grafana alerting capabilities. Those are a little bit limited compared to Prometheus and Alertmanager, but flexible enough to fulfill our requirements.

All those integrations are still open questions, as they should be more analyzed and evaluated.

3.4.6 Summary

To summarize, we chose the following stack of technologies to implement our monitoring support for MANTA Flow Agent:

- Java Agent - to enable monitoring capabilities (collecting and sending metrics data to MANTA Admin GUI) for our applications by attaching it to them at their startup.
- MXBeans from JMX - to gather metrics from JVM.
- OSHI library - to collect system and process metrics.
- Byte Buddy - to implement custom monitoring API for defining and exposing user-defined metrics.
- Prometheus - as a time-series database, primarily for storing and querying metrics data.

- Spring Boot - as an application framework to build monitoring Java Agent.
- JMS - messaging for sending metrics data to MANTA Admin GUI.

The design and implementation details of our provided solution are in the following chapters.

4. Design

In this chapter, we describe the final design for monitoring support for MANTA Flow Agent, individual aspects of the architecture, what decisions had to be made when we were considering multiple alternatives.

As we mentioned in the previous chapter, we designed and built a monitoring solution based mainly on custom monitoring Java Agent and Prometheus, time series database. A high-level overview of the final architecture can be seen in the following Figure 4.1. We will discuss about its components in more detail in the following sections.

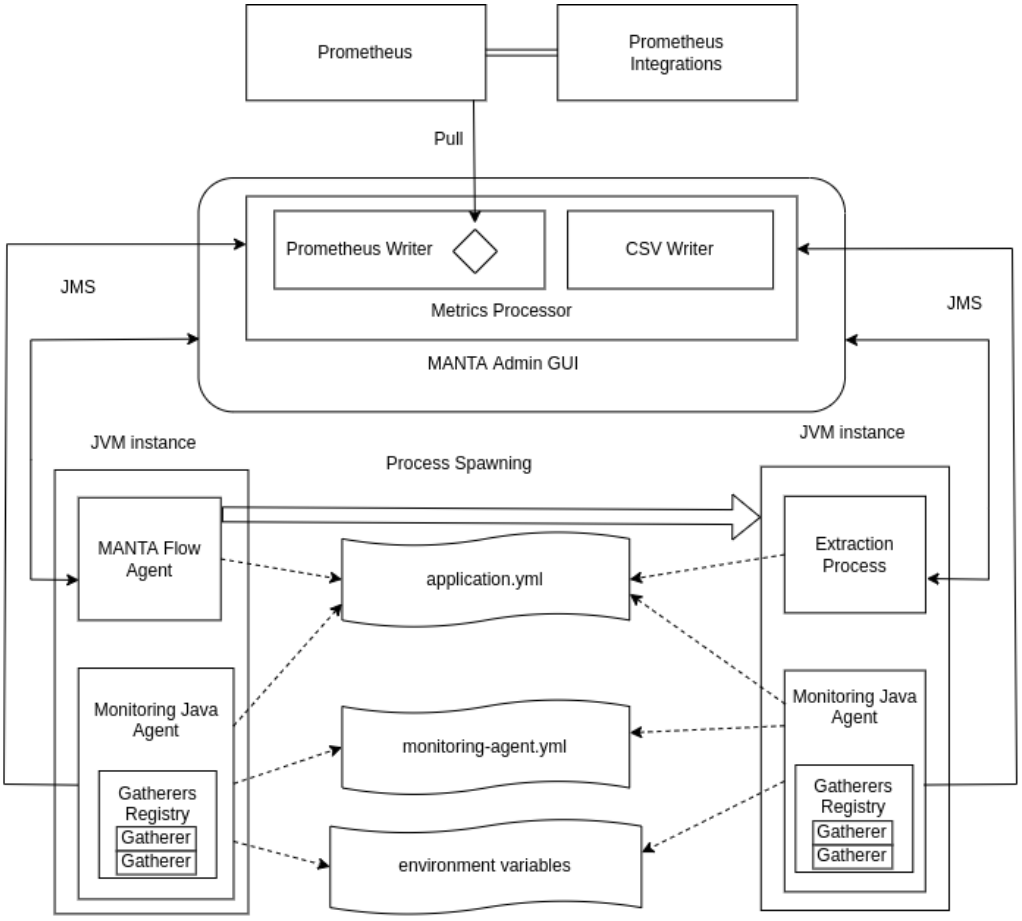

Figure 4.1: Final Design

4.1 Monitoring Java Agent

To gather metrics data, the monitoring Java agent is used. It is also responsible for transporting such data to MANTA Admin GUI. Our monitoring targets are MANTA Flow Agent and its spawn extraction processes. They run in their own JVM. We decided to use a strategy where the Java agent's jar file is passed as an argument when the application is starting, so the Java agent starts before the actual application. The alternative is to start the Java agent after the application has started. E.g. the user could somehow trigger that the monitoring

agent should start. But from the implementation point of view, it's not often possible, as in the documentation of package *java.lang.instrument* it is stated that an implementation for package *java.lang.instrument* **may** provide a mechanism to start the Java agent after the application. We found out, that the Java implementation from Oracle supports that option, but in general, it's a JVM vendor-specific if this feature is supported and how, including such API. We decided that both MANTA Flow Agent and extraction processes should be started with enabled or disabled monitoring Java agent. For extraction processes, there is an option within Admin GUI to toggle this functionality, for MANTA Flow Agent, the desired startup script has to be selected to start MANTA Flow Agent with enabled or disabled monitoring Java agent.

We will discuss, that the monitoring agent is also able to do instrumentation, in our case to implement monitoring API. We want to use the Load-Time instrumentation, instrumentation when a class file is instrumented while it is loaded by the JVM. That process is however irreversible. There are also some alternatives. One option is, that before the instrumentation of some class, we could decide if we want to do this specific instrumentation, e.g. via some configurations. Alternatively, the class is instrumented always, if a class should be instrumented, but inserted code could have a logic that could enable or disable inserted functionality. We chose the simplest, less complex way and we do not provide any option to enable or disable the inserted functionality on the fly. However, we leave some space for potential implementation for enabling and disabling specific instrumentation on the application start.

We wanted to keep the configuration as simple as possible. To use the monitoring agent, some necessary parameters have to be provided. Those are related to communication, concretely the JMS broker hostname, port and an optional keystore configuration for mTLS. All those are shared from the configuration file for MANTA Flow Agent (*application.yml*). The last important shared configuration value is *Agent ID*, which is used to identify the source of metrics data. We provide also a configuration file only for the monitoring agent (*monitoring-agent.yml*). In the initial phase, we provide only a configurable option for the *scraping interval*, which determines how often metrics data should be gathered and sent to MANTA Admin GUI.

In the next Figure 4.2, we can see what is happening within the premain function, function responsible for initialization of monitoring Java agent. After the premain method execution, the original application is started. Most of the configuration loading is handled by the Spring framework. During the implementation, we encountered some problems with class loading. We resolved that by providing only light monitoring agent which on its startup bootstraps the real monitoring agent implementation. More details about why we had to do this and how it is done, can be found in the next chapter.

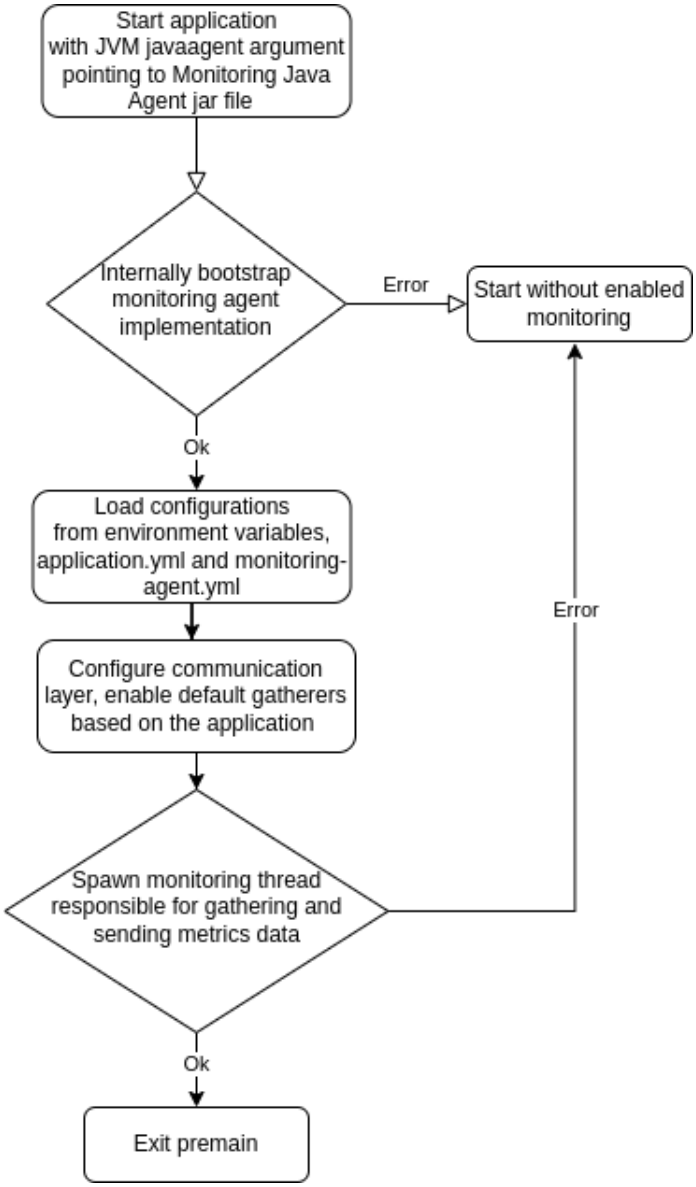

Figure 4.2: Premain Sequence

4.2 Gathering Model

To implement metrics data gathering, we introduce a *gatherer*. It is a component that gathers and potentially pre-computes some metrics data. Gatherers are part of the monitoring Java agent, they are registered in *gatherers registry*. Different gatherers could gather different metrics. The main purpose is to provide some measure. In general, we can recognize two types of gatherers depending on how they update their state:

1. A gatherer update is made asynchronous from other sources, components, so the gatherer doesn't know when it happens. This type of gatherer could be used to implement some sort of accounting.

2. A gatherer actively fetches or computes metrics from some sources, e.g from the system where the application is deployed. As this gathering consumes

time and it can be somehow cheaper or more expensive, we could distinguish between two categories: long-running gatherings and short-running gatherings.

The first type of gatherer we will use in the implementation of monitoring API. There, the user code updates the gatherer state with the up-to-date values. The second type is suitable e.g. for JVM metrics, where JVM metrics are pooled periodically from JVM. We also try to avoid long-running gatherings, while such gatherings potentially could block gathering for a long time.

To support enabling and disabling gatherers we could consider two strategies. One is allowing a selection of which gatherers should be in action when starting the application and do not consider enabling/disabling gatherers in runtime, or make this more flexible and allow enabling/disabling gatherers dynamically during the whole runtime.

Alternatively, we could start MANTA Flow Agent with all suitable gatherers enabled by default and make it possible on startup to blacklist some of them and also provide an option for remote disabling. Some complications could arise when considering disabling gatherers which use some sort of instrumentation as complete rollback couldn't be easily achievable. Turning off some gatherers could be useful if we find out that they are so much affecting performance.

As we said in the previous section, we decided to implement the option when all gatherers are enabled by default and make it possible to implement gatherers disabling by blacklisting them on the application startup in the further development iterations.

Gatherer design alternatives

The gatherer has to interact with the transport layer. There are 2 options for how this transport component could work:

(i.) Transport component actively, periodically, reads a gatherer output and transports it to the remote side.

(ii.) Transport component fetches values from a gatherer only on the demand when the command for the metrics collection arrives from the remote side.

In both cases, the transport layer could also forwards data for some adapters, to make retrieved data available by some different channels, e.g. via JMX or simply write data into a file system.

The following three main strategies for gatherers concerning the data gathering and transporting responsibility have been analyzed:

1. Gatherer prepares values for some transport component which reads those values. There are two options:

   (a) Gatherer as a self living unit. A gatherer lives as a thread and gathers metrics and prepares metrics data for a potential read by some transport component.

(b) Gatherers are in one group where one gathering control thread calls gatherers to retrieve their values and save them, prepare them to be read by a transport component. The whole work of gatherers is done within such calls.

In both cases, there is a transport component, which can retrieve those metrics data and that component must read them before forwarding them to the remote side.

This strategy, when the gatherer prepares values for fetching, has some decision points, such as if only the last measure should be saved or the last n measures.

2. Gatherer directly interacts with a transport component. That means, that a gatherer know how to e.g. send metrics data.

   Similar to the previous strategy there are two options for how a gatherer could live:

   (a) Gatherer as a self living unit. A gatherer lives as a thread and a gatherer interacts with a transport component.

   (b) Gatherers are in one group where one gathering control thread calls gatherers to retrieve their values and interacts with the transport component within those calls.

   We prefer interaction with the transport component to be asynchronous and this transport component should send metrics data transparently.

   Again, as in the previous strategy 1, both approaches i and ii for a transport component can be used. For the ii, it is the question, what to do, if no command for fetching data doesn't arrive, e.g if such metrics data should be queued and so on.

3. Transport component picks metrics from gatherers.

   In this strategy, the transport component has also the role of a gathering control thread in alternative 1b.

   Both approaches (i and ii) can be used.

In all three gatherer design alternatives both alternatives i and ii for transporting metrics data could be implemented.

We agreed on the use of the approach 3. The first suggestion has the disadvantage of creating other unnecessary communication channels between gatherers and the communication layer, e.g. via a data queue. The second alternative makes gatherers aware of some communication layer, so such gatherers would have to implement what the *send* means and in general, we would tightly couple two independent responsibilities. The third one, the selected one, the big disadvantage is that the main control thread could be blocked when retrieving metrics from a gatherer for an unknown time. We do not provide any mechanism to avoid such situations. For our gatherer implementations, this could be less probable that this could happen, as we have the source code of such gatherer implementations

under control. Problematic could be the monitoring API usage, as the user could define a gatherer which could block data metrics retrieving in an uncontrolled manner. One simple way how to resolve such a special case is that we could separate retrieving of the user metrics from the retrieving of metrics data from our default gatherers.

For the transport component, the alternative i have been chosen over alternative ii. Approach i is similar to the Push model and ii is more closer to the Pull model used in other monitoring solutions. Approach i makes getting metrics data into the remote side simpler, as one-way messaging can be used and it can be implemented as stateless. However, this model could bring some disadvantages as it is generally described in the Push model, see Section 3.2.1.

The final gathering model in our monitoring Java agent looks like this: on applications startup, all default gatherers are enabled and registered into the gatherers registry. On runtime, some other gatherer could be dynamically registered into the gathers registry as well. This dynamically created gatherer is created only when the monitoring API is used, more details are in Section 4.4. We periodically iterate over all registered gatherers and retrieve their metrics data within one control daemon thread. When we got all metrics data, we send them in one message to the remote side. That all is repeated after the scraping time has elapsed, after asynchronously submitting metrics data to the transport layer. Once a gatherer is registered, then it cannot be unregistered. This whole model is shown in Figure 4.3:

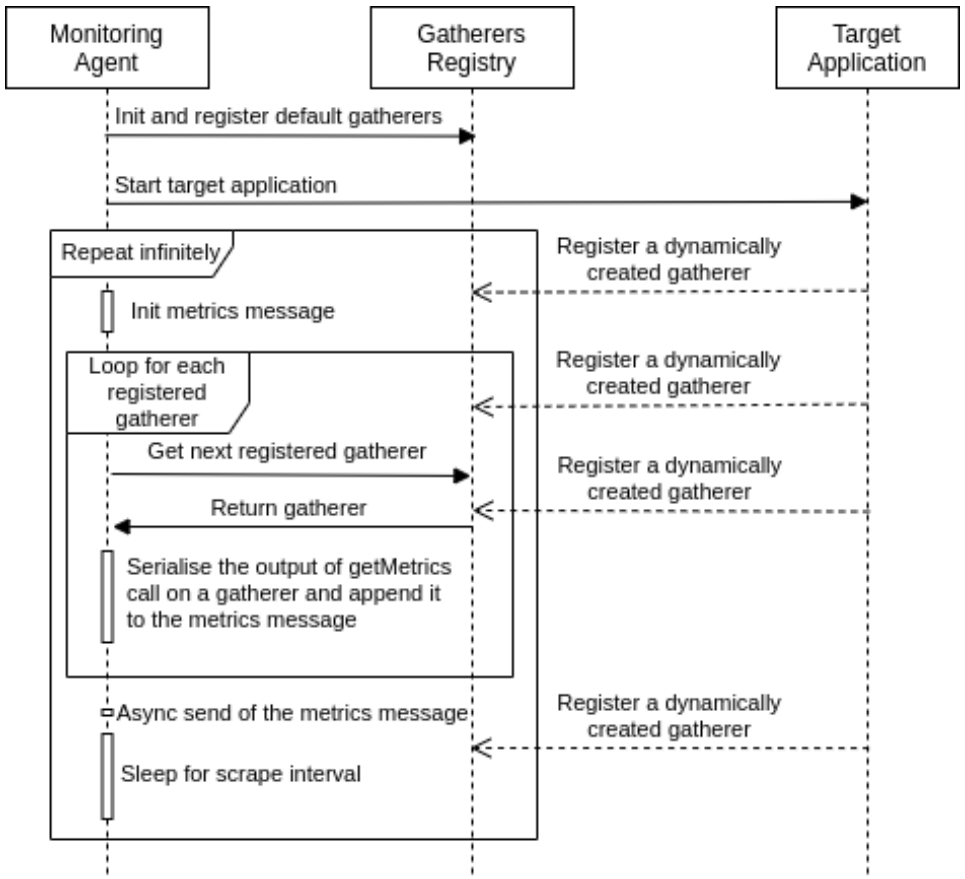

Figure 4.3: Gathering Model

The gatherer can be any class that implements *Gatherer* interface from the Listing 4.1. Its essential method, *getMetrics*, is called regularly only when such Gatherer is registered into the global gatherers registry.

```
public interface Gatherer {

  /*
   * @return Null if no data available/provided
   *              else current metrics data.
   */
  Metrics getMetrics();

}
```

Listing 4.1: Gatherer interface

There are two ways how gatherers can be registered. One way is the static registration, on the monitoring agent initialization. As we use Spring, we decided to use Spring-managed beans, which can be used to create gatherer objects via the functional interface *GathererProvider*, as it is shown in Listing 4.2. In such a way, gatherers can be collected from the enabled gatherer provider beans. The second type is for dynamically registered gatherers, created for every user type that use our monitoring annotation-based API, see Section 4.4. For every such annotation an *Annotation Installation* object is required to create, see Listing 4.3. This object is responsible for binding interceptor implementation for specific method annotation. All its constructor parameters are managed beans. When a type that contains an annotated method with the target annotation is loaded for the first time by JVM, then at that moment the interceptor object for that type is created. This interceptor is also a Gatherer and is registered into the global registry, more details about interceptor objects can be found in Section 4.4.

```
@Service
public class RandomValuesGatherer implements GathererProvider {

  @Override
  public Gatherer getGatherer() {
    return () -> {
      Metrics metrics = new Metrics("random");
      metrics.set(Math.random(), "value");
      return metrics;
    };
  }

}
```

Listing 4.2: A gatherer implementation

In the next Listing 4.3, we can see the installation class for the monitoring method annotation @TimeIt. We do not show here details for its interceptor implementation. Interceptors are described in more detail in Section 4.4.

The relationships between @Services classes, monitoring annotations and interceptors classes are shown in the diagram in Figure 4.4:

```java
/**
 * Installation for annotation TimeIt
 */
@Service
public class TimeItInstallation {

  public TimeItInstallation(GatherersRegistryProvider
      registryProvider, Instrumentation instrumentation) {
    ...
  }

}
```

Listing 4.3: An annotation installation

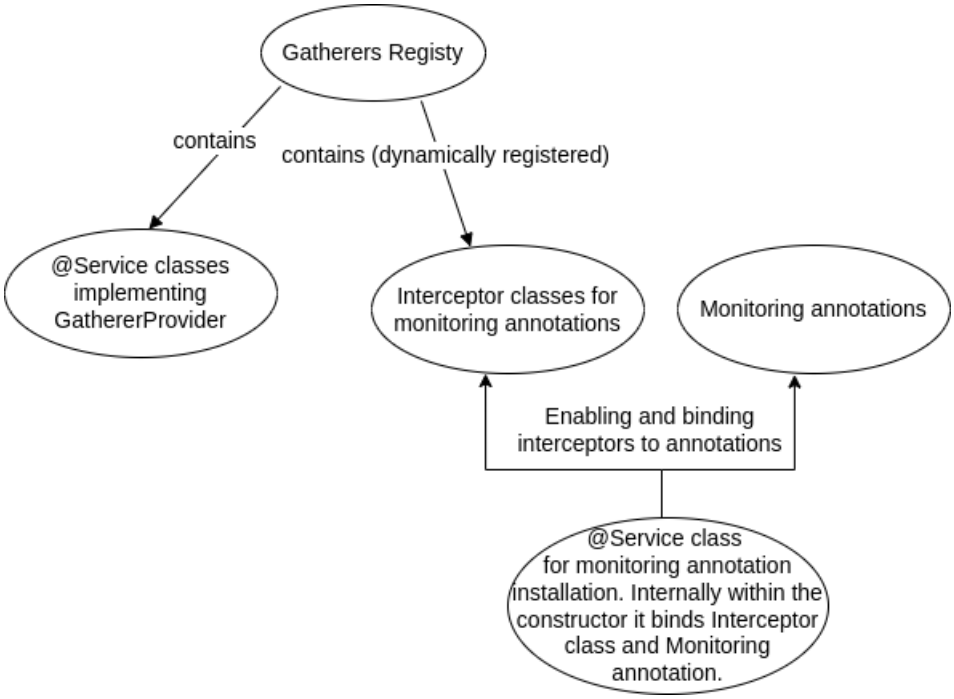

Figure 4.4: Relationships between @Services classes, monitoring annotations and interceptors

Finally, in Figure 4.5 we can see what kind of metrics a gatherer implementation can collect:

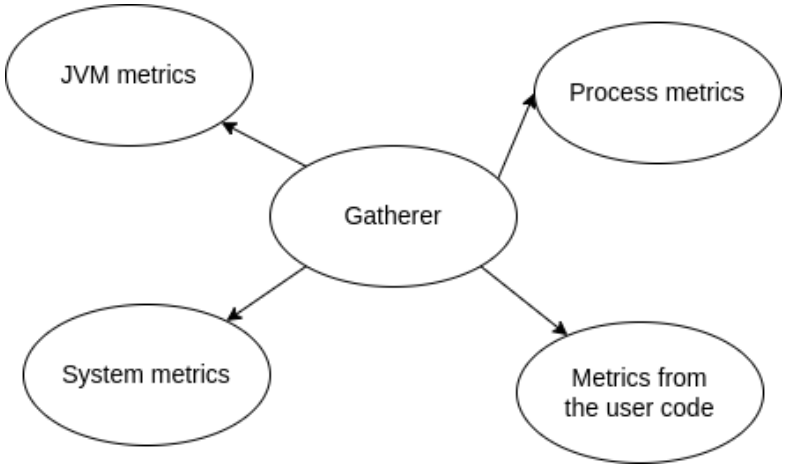

Figure 4.5: Gatherer's target metrics types

4.3 Metrics Data Format and Transporting

After metrics data are periodically gathered and collected from the target application which we want to monitor, then they have to be sent to MANTA Admin GUI. To transport metrics data from such applications, MANTA Flow Agent and its extraction processes, we decided to use one-way messaging. Those messages are periodically sent after the application is started. This interval is configurable via MANTA Admin GUI for extraction processes and for MANTA Flow Agent, the configuration file (*monitoring-agent.yml*) mentioned in Section 4.1 can be used.

As the main application framework for our monitoring Java agent is the Spring Boot framework, we prefer to use its support for JMS messaging instead of the direct usage of the raw JMS API. JMS messaging is also used across the MANTA Platform and MANTA provides for developers a module to make working with JMS messaging even easier.

Our monitoring Java agent takes connection configuration from the configuration for MANTA Flow Agent, from *application.yml*. To connect to the JMS broker, it's necessary to know the broker hostname, port, and optional keystore configuration if mTLS is enforced. To identify the source of metrics message, an Agent ID is used, which can be also found in that configuration file.

JMS supports two messaging models. One is the point-to-point model and the second one is the publisher-subscriber model. In our case, the point-to-point model fits better as we want to have only one component to consume metrics data, which is placed in MANTA Admin GUI. However, switching to the publisher-subscriber model can be done easily. JMS uses terminology JMS Topics and JMS Queues for the publisher-subscriber model and the point-to-point model respectively. In that terminology, we use JMS Queues and one such queue is for us enough. It's name is *queue/MonitoringMetrics*.

The data model for internal representation for metrics created from gatherers is the following. The metric has a name, we prefer snake case, 64bit double value and an optional set of labels. Such a label consists of a label name and label value, both are text values. So, we stick to Prometheus data model, see Listing 3.4. As we do not want to support events logging, numeric type is sufficient for

our metrics. Labels could be used to add additional information for a metric, e.g. a phase of a process in which was that metrics collected. Serialized internal representation for metrics data can be seen in Listing 4.4.

In general, a JMS message consists of headers, properties and a body. The body could be one of the following types: StreamMessage, MapMessage, TextMessage, ObjectMessage, BytesMessage or XMLMessage type. We decided to use TextMessage type for the body of metrics messages. Then metrics data from the gatherer has to be serialized into text form, to be passed into TextMessage. Information about the type of such encoding is the part of metrics message, it is in the message header. That header is used to determine which decoder should be used to decode, deserialize, the delivered metrics data in text form.

For metrics data transport format we chose JSON. We could use more sophisticated format, like Protocol Buffers, but we are not dealing with the huge amount of data and high data throughput, so JSON parsing shouldn't cause such a slowdown. However, the implementation should be prepared to change the metrics format used for serialization and deserialization. The example of metrics data in transport format is shown in the following Listing 4.4:

```
[
    {
        "name":"jmx_os_processors",
        "value":8,
        "labels":{}
    },
    {
        "name":"jmx_os_info",
        "value":1,
        "labels":{
            "name":"Linux",
            "arch":"amd64",
            "version":"5.15.6-arch2-1"
        }
    },
    {
        "name":"network_sent_bytes_total",
        "value":1.33896017E9,
        "labels":{
            "interface":"wlp1s0"
        }
    }
]
```

Listing 4.4: Metrics written in transport JSON format

As we said, we use the JMS header to determine which encoder should be used to deserialized delivered metrics data. In the previous Listing 4.4 we can see, that it doesn't contain any timestamps for metrics data. Gatherers do not handle timestamps for collected metrics values. The timestamp is appended to the JMS message, as a header. This timestamp value contains the timestamp when the message was sent, concretely when the control thread collected all metrics from gatherers and submit that data to the transport layer. The disadvantage of this approach is, that we do not know when exactly the metric was generated.

The advantage is, that gatherers do not have to deal with timestamps at all. For monitoring purposes, the precise time of the collected metric is not needed.

We prefer the use of JMS headers instead of JMS properties, as they could be used in the listener methods on the remote side as a selector for receiving specific messages, which is Spring specific feature. This can be used to distinguish the different application types from which the message was delivered, e.g if the message is from the extraction application spawned by MANTA Flow Agent or directly from the MANTA Flow Agent application. Also, a JMS header could be used for transporting other metadata not directly related to the metrics. For messages from extraction processes those metadata are: *AGENT_ID*, *WORKFLOW_EXECUTION_ID*, *SCENARIO_NAME*, *CONNECTION_ID* and *TECHNICAL_CONNECTION_ID*. The value of AGENT_ID is obtained from the configuration file of MANTA Flow Agent and the others are obtained from the extraction command, delivered to MANTA Flow Agent, sent by MANTA Admin GUI. In metrics messages from MANTA Flow Agent, we require only AGENT_ID metadata header to be present. Those metadata helps us to identify the source of metrics messages and add the context for such metrics messages. Those metadata can be optionally used to expand the original label sets of metrics data in delivered metrics messages. The content of the JMS messages is described in the schema in the following Figure 4.6:

Metrics Message for Extraction Application	
JMS_DESTINATION	queue/MonitoringMetrics
JMS_TIMESTAMP	1650289171
JMS_TYPE	TextMessage
agentId	Agent[6f405906-b139-4a0d-9ab1-e818f032cf0c]
messageType	ExtractorAppMetricsMessage
MANTA_WORKFLOW_EXECUTION_ID	42
MANTA_SCENARIO_NAME	postgresqlExtractorScenario
MANTA_CONNECTION_ID	Postgresql DEV1
MANTA_TECHNICAL_CONNECTION_ID	my-dictionary-id-postgreSQL
MetricsBodyFormatDTOVersion	JSON_V1
body	*JSON encoded metrics data*

Metrics Message for MANTA Flow Agent	
JMS_DESTINATION	queue/MonitoringMetrics
JMS_TIMESTAMP	1650290142
JMS_TYPE	TextMessage
agentId	Agent[6f405906-b139-4a0d-9ab1-e818f032cf0c]
messageType	MantaAgentAppMetricsMessage
MetricsBodyFormatDTOVersion	JSON_V1
body	*JSON encoded metrics data*

Figure 4.6: JMS Metrics Messages

On the monitoring agent startup, it's determined which kind of JMS metrics messages will be used during the application execution. The specialized implementation responsible for creating the right type of messages is instantiated as a Spring bean. This creator then fills JMS messages with the appropriate headers for the application type. The relationships between different types of messages and messages creators at the class level are shown in the following Figure 4.7.

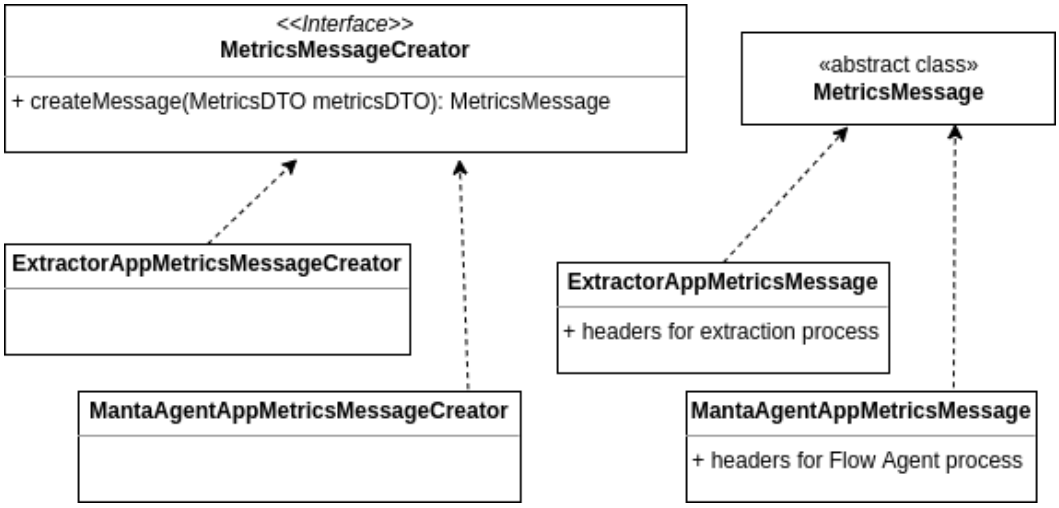

Figure 4.7: JMS Messages Creators

4.4 Monitoring API

The part of monitoring support for MANTA Flow Agent is also the monitoring API. With this API, the developer could instrument an application to generate some metrics data. In our case, the target applications are MANTA Flow Agent and its extraction processes.

This API is annotation-based and supports two categories of metrics. We have some predefined annotations which enable collecting some specific metrics, but we also provide a generic annotation that can be used to export any user-defined metrics.

Those annotations are intended to be used on instance methods. To use them, one has to annotate a class with the annotation *@Monitor*. Then, the annotated methods will be enhanced with the metrics collecting functionality described in the following paragraphs. Only one monitoring annotation can be attached to the instance method. When they are used, the desired method call metrics will be collected. The name of such metrics starts with the prefix *annotation_*, followed by the name of annotation and user-defined metric name.

4.4.1 Annotations with specific monitoring sematic

When we use this type of annotation, we are interested in some specific aspect of the original method call. Those annotations are:

- *@TimeIt* - exports time-related metrics to the annotated method call, such as how much time was spent in this function, when the last method call started and finished, how many times it was called.
- *@TrackEnumReturnValue* - tracks how many times the method was called and counts how many times and which enum value across the whole range of enum values was returned.
- *@ExceptionMetered* - tracks which types of exceptions and how many exceptions the method call threw.

- *@TrackEnumArgument* - tracks how many times the method was called with the enum value. This tracks the values only on the first argument which matches the TrackEnumArgument argument defining in which enum type of argument we are interested.

In the following Listing 4.5 we can see how those annotations like @TimeIt, @ExceptionMetered and @TrackEnumArgument could be used in practice:

```
@Monitor
class PostgreSQLExtractor implements Extractor {

  @TimeIt
  @Override
  public void extract() throws ExtractionError {
    ...
    extractImpl();
    ...
  }

  @ExceptionMetered
  private void extractImpl() throws ExtractionError {
    ...
    for (Table table : tables)
      extractTable(table);
    ...
  }

  @TimeIt
  private Metadata extractTable(Table table) {
    ...
    for (DdlType ddlType: extractDdlTypes(table)) {
      List<EntityEntry> entities = getEntitiesForDdlType(ddlType,
          schemas);
        ...
    }
    ...
  }

  @TrackEnumArgument( enumToTrack = DdlType.class )
  private List<EntityEntry> getEntitiesForDdlType(DdlType ddlType,
      Collection<Schema> schemas) {
    ...
  }

  ...
}
```

Listing 4.5: The use of monitoring annotations

4.4.2 Annotation to expose custom metrics

The developer can expose custom metrics from the code via method annotation *@MetricsProvider*. This annotation has to be attached to the method which returns an instance of *Metrics* class. Same as in the previous annotations, a

class containing such annotated methods has to be annotated with the class annotation @Monitor to make method annotation effective. To register that method into the metrics gathering mechanism, one has to instantiate the class which contains such annotated method. Then, this method will be called regularly (see scraping interval in Section 4.1) to collect recorded metrics. In the body of such annotated method, it is recommended to only wrap metrics data into an instance of the Metrics class and don't do any time-consuming operations. It's up to the developer to continuously update the state of exported metrics data.

Custom metrics are recorded into an instance of the *Metrics* class. With this class, it is possible to expose 64bit numeric metrics value with its name and the set of labels. Those labels should be primarily used to define the dimension on which the multidimensional queries could be executed, e.g. to identify the source of metrics or differentiate request types or stages. However, the space of labels values should be bounded and they shouldn't be used to track some frequently changing values. In other words, the user should avoid using labels to store dimensions with high cardinality.

In the following Listing 4.6, we can see how such API could be used to expose some extractions statistics while the application is running:

```
@Monitor
class ExtractionStatistics {

  @MetricsProvider
  private Metrics exposeMetrics() {
    Metrics metrics = new Metrics("extractions");

    metrics
     .set(assetsExtracted, "assets", "extracted", "total")
     .set(assetsPending, "assets", "pending", "count")
     ;

    for (Map.Entry<String,String> durations : stagesDurations.
        entrySet()) {
       String stage = durations.getKey();
       String duration = durations.getValue();
       metrics
        .withLabel("stage", stage )
        .set(duration, "stage", "duration", "seconds")
        ;
    }

    metrics
     .withLabel("version",  BUILD_VERSION)
     .withLabel("license",  getLicenseMode())
     .set(1, "info")
     ;

    return metrics;
  }

  ...
}
```

Listing 4.6: The use of @MetricsProvider annotation

4.4.3 Annotations and Interceptors

Introduced monitoring annotations are method annotations. To implement their functionality, we use Byte Buddy, a code generation library (see Section 3.1.6). With this library, we can modify Java classes during the runtime of a Java application and without the help of a compiler.

A monitoring aspect is represented by some interceptor class. The instance of an interceptor can enhance the original method functionality. In our case, it usually does some measure on the original method call. Interceptor is also a gatherer and it can be registered to the gatherers registry.

An interceptor object is created for every user type which uses the monitoring annotation associated with this interceptor. To register such dynamically created interceptor to gatherers registry, an annotation installation object is used. It is also responsible to register a class transformation, which modifies the original method, that all with Byte Buddy's help.

To summarize, on the code level, for every monitoring annotation we also have an interceptor class. There is also a dedicated installation class for such a combination that connects those entities. The instance of such class registers the action that creates the instance of interceptor and registers it to the gatherers registry. This action is taken only on types that contain the monitoring annotation, and when such types are loaded by JVM. All those relationships are shown in Figure 4.8. More implementation details can be found in Section 5.1.3.

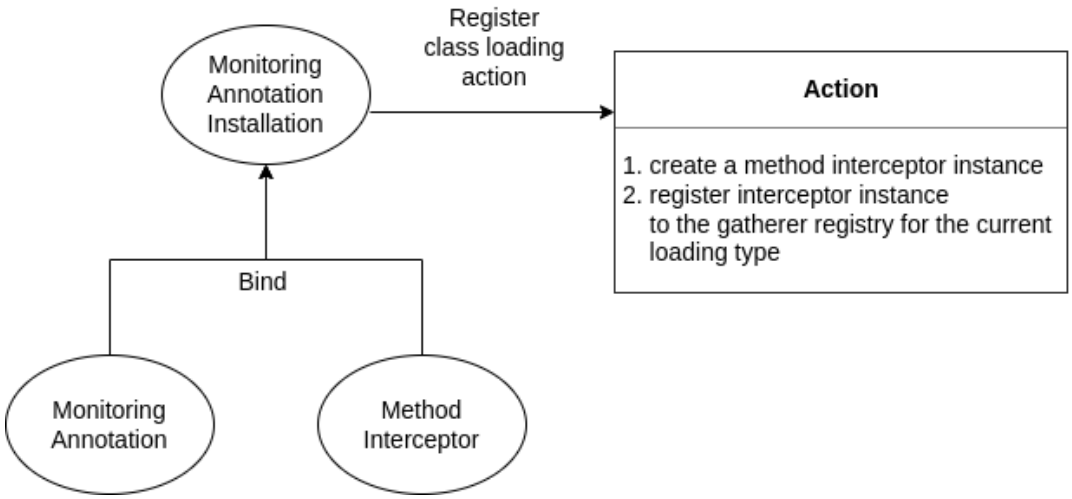

Figure 4.8: Annotation Installation

4.5 Metrics Data Integrations

In the previous sections, we talked about how metrics data are gathered and transported from the monitoring Java agent to MANTA Admin GUI. The main consumer of such metrics data is part of MANTA Admin GUI.

On the MANTA Admin GUI side, there are listeners for metrics messages. Those messages are forwarded to a consumer. There are usually metrics messages deserialized based on the encoding header value and published, e.g. into the file system or Prometheus.

We have to be aware of different scenarios of how MANTA as a product could be deployed, e.g. from a compact solution to highly scalable deployments. For us it means, that we have to be ready to simply switch the way how and where metrics data are stored.

As a result, we suggest using Pushgateway from the Prometheus stack (see Figure 3.7) to propagate metrics data to Prometheus. This service listens for metrics data. We can push metrics data from listeners directly to the Pushgateway. Then, Prometheus can be configured to pull those metrics from such Pushgateway instance.

For compact deployments, we suggest writing metrics data directly into the file system. We chose CSV format as it's in both cases, machine and human readable enough for our needs. Metrics data in CSV format are shown in Listing 4.7. The first column is a timestamp of gathered metrics with millisecond precision. The timestamp column is followed by metrics columns which can optionally contain labels set.

```
ts,cpu_load_average_1m,cpu_ms[cpu=0|mode=user],cpu_freq_hz[cpu=0]
1647434548477,2.42,538360,2394947000
1647434551996,2.63,538000,2531826000
1647434555635,2.82,537390,2716053000
1647434559090,2.63,536650,2716575000
1647434562519,3.32,535420,2799157000
1647434565906,2.67,535050,2806688000
1647434569307,3.34,534780,2843376000
1647434572712,2.81,534610,2852152000
1647434576123,2.67,534300,2936554000
1647434579511,3.34,533330,3003000000
```

Listing 4.7: Metrics data in CSV format

As we can fill metrics data transferred from the monitoring agent into the time series database Prometheus, other advanced integrations could be easily introduced. For simple metrics visualization, the graph plotting functionality of the embedded web application of Prometheus could be used. Alternatively, Grafana (see Section 3.8), a widespread and advanced visualization platform, could be used. Similar holds for the alerting integrations, as Grafana is capable of that, or Alertmanager from Prometheus stack could be used. The final suggested metrics data integrations can be seen in the following Figure 4.9:

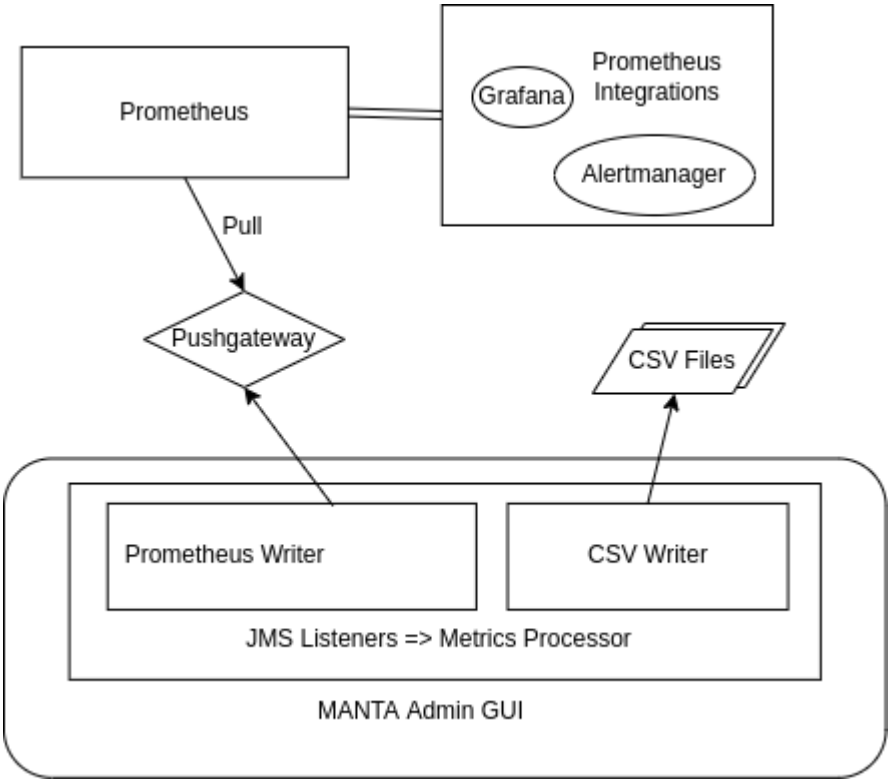

Figure 4.9: Metrics Data Integrations

5. Implementation Details

In this chapter, we provide more details about gatherers implementations and the integrations of the output of the Java monitoring agent within MANTA Admin GUI.

5.1 Metrics Gathering

As the result of the analysis of the functional requirements, we want to gather metrics from the running JVM, metrics from the system where MANTA Flow Agent is deployed, metrics related to the JVM process from the OS perspective, and metrics exposed by the monitoring API. In the next sections we describe how we achieved the gathering of such metrics.

5.1.1 JVM metrics

To gather metrics related mainly to the running JVM we implemented Gatherer interface around MXBeans instances. From the following MXBeans we are able to collect the listed metrics:

- *RuntimeMXBean* - JVM information like JVM vendor, name, version, uptime and the start time of the JVM.
- *OperatingSystemMXBean* - operating system's basic information like the OS name, OS version and OS architecture. It exports also the number of available processors to the JVM and the system load average for the last minute.
- *MemoryMXBean* - heap and non-heap memory usage like the amount of memory that is committed for the JVM to use, the maximum amount of memory that can be used for memory management, the amount of used memory.
- *GarbageCollectorMXBean* - the number of collections and the approximate accumulated collection time for the specific types of garbage collectors.
- *ClassLoadingMXBean* - the total number of loaded classes into JVM, the number of currently loaded classes in JVM, the total number of classes unloaded since the JVM has started execution.
- *ThreadMXBean* - information about the thread system of the JVM like the current thread count, the current number of the daemon threads, the peak live thread count, and the total number of the started threads.

To implement Gatherers wrapping MXBeans, we have to be aware of the right package for MXBeans interfaces. The JVM platform-independent interfaces are in the package *java.lang.management*. We also encountered situations when the provided value from MXBean wasn't meaningful at all, e.g. it was always -1 on the Windows platform, as it is a valid value from unsupported platforms.

To partially resolve it, we use JVM vendor-specific interfaces (from the package *com.sun.management*) which provide alternative functions to retrieve desired values. However, to access such interfaces in the code, we do not directly import those interfaces from a vendor package but we use the *MBeanServer* ability to dynamically call those JVM vendor-specific interfaces to retrieve desired metrics. Metrics gathered from the JMX system have prefix *jmx* in their metric name.

5.1.2 System and Current Process Metrics

To get metrics from the system where MANTA Flow Agent is running and from the current process we use the library OSHI (see Section 3.1.8). This library is capable to retrieve the following system metrics:

- CPU metrics: the number of physical CPU packages/sockets in the system, the number of logical and physical CPUs available for processing, the CPU signature, the maximum frequency and the current frequency of the logical CPU, the system load average (if available) for 1, 5, and 15 minutes. For every logical CPU we track time spent in User, Nice, System, Idle, IOwait, Hardware interrupts, Software interrupts/DPC and Steal state. For all those states also a system-wide CPU spent time is provided.
- RAM metrics: the amount of actual physical memory and the amount of currently available physical memory.
- disk metrics: the number of read and written bytes from the disk, the number of reads and writes from the disk, the time spent reading or writing, disk size.
- file system information: the total space and usable space for every mounted volume.
- network metrics: received and sent bytes for every network interface, IPv4 and IPv6 addresses for every network interface.
- OS information: the OS manufacturer, family and bitness, the time of boot and uptime, the number of currently running threads and processes, the hostname.

With the use of the OSHI library, we also retrieve the following metrics related to the currently running process:

- disk metrics: the number of read and written bytes.
- RAM metrics: the Resident Set Size and the Virtual Memory Size.
- file system information: the number of opened files.
- process information: the process ID, the process start time, the user time and kernel time used by the process, the number of threads being executed by the current process.

- network information: connections attributes for the current process network connections like the local port, the foreign port, the type of connection (e.g. TCPv4, UDPv6) and the state (for TCP only).

For every category of metrics and the source (system/the current process) we implement the Gatherer interface to wrap method calls on OSHI objects to retrieve those metrics. To distinguish category we use prefixes in their metrics names such *cpu*, *disk*, *fs*, *mem*, *os*, *network*. Metrics related to the current process contain prefix *process* in their metric name.

5.1.3 Metrics from the Monitoring API

We implemented the monitoring annotation-based API which can be used to track specific aspects on the instance methods or to expose user-defined metrics. To track a specific aspect of the method call, developer has to annotate the method with the specific method annotation. To expose user-defined metrics, the developer has to annotate the method(s) responsible for exporting user-defined metrics with the method annotation. In both cases, classes containing such method annotations have to be annotated with the class annotation *Monitor*. This marker annotation serves for the user to indicate the presence of the monitoring method annotations within the class and also to speed up and simplify inserting the monitoring aspect into the target class. For inserting the monitoring aspect we use Byte Buddy's *AgentBuilder* which can be used to register class transformation. The marker class annotation Monitor is used for filtering classes that can be potentially modified.

As a proof of concept we implemented the following annotations to track the specific method call metrics: *TimeIt*, *TrackEnumReturnValue*, *ExceptionMetered* and *TrackEnumArgument*. More information about their semantics can be found in Section 4.4.1. Metrics exported by those annotations have metric name prefix *annotation*.

To export custom metrics, the user can use method annotation @MetricsProvider on a method returning our *Metrics* class. More information about the usage can be found in Section 4.4.2. Metrics exported by this annotation have metric name prefix *custom*.

For all monitoring annotations it holds, that they have the installation class, which instance is responsible for registering their class transformation logics. Within such transformation, which is triggered on the class loading into JVM, an instance of the interceptor, specific for annotation, is created. This interceptor is also Gatherer which is registered into the gatherers registry. More details are described in Section 4.4.3.

For MetricsProvider method annotation, its interceptor's method *afterConstructor* is called when an instance of the type, which uses MetricsProvider annotation, is constructed. Within this method, the reflection is used to find all methods on the current type which return Metrics class and do not take any arguments. For every such instance method a Gatherer instance is created, which only calls such method to retrieve metrics, and this gatherer is registered to the gatherers registry. This flow can be seen in Figure 5.1. Interactions between gatherers registry and thus dynamically created gatherers are shown in Figure 4.3.

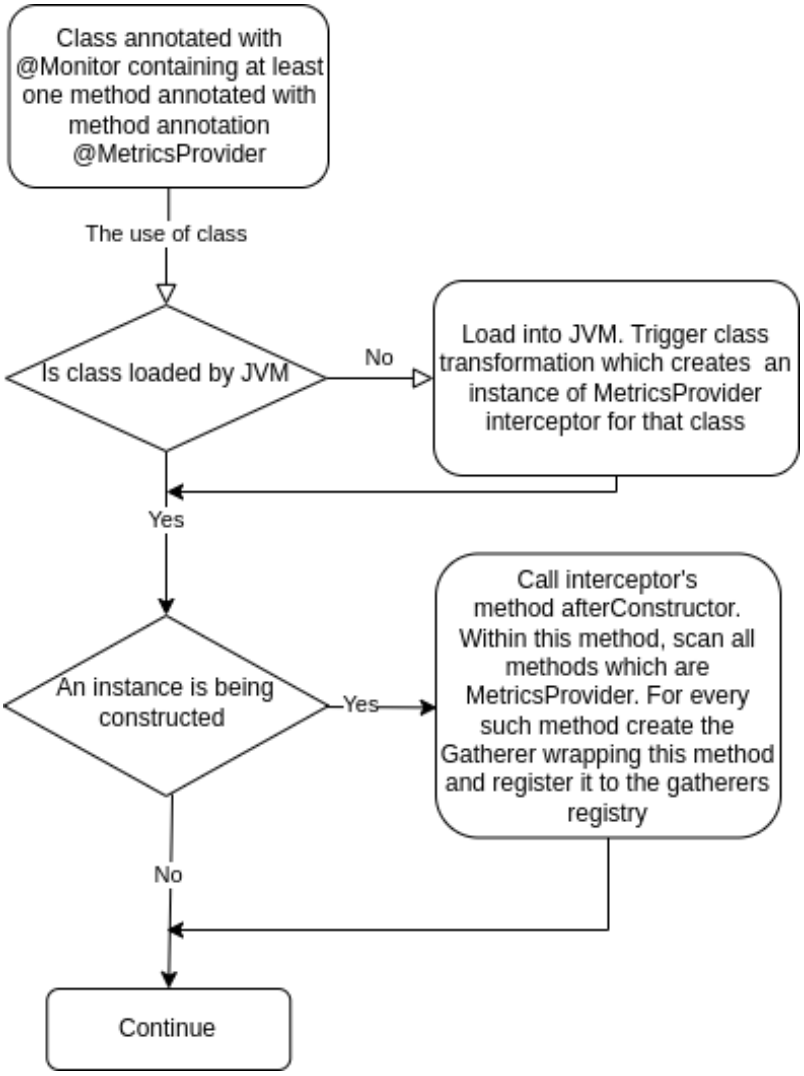

Figure 5.1: Creating an instance containing @MetricsProvider method

For other monitoring method annotations, the interceptor specific for such annotation is instantiated for every type which uses such method annotation. The original method is replaced with the method provided by the interceptor. This interceptor contains a map of objects (of type for which the interceptor was created) to method call dispatchers. Method call dispatcher has a map of the original method names and references to their implementation. When an intercepted (annotated) method is called, then it goes through the double dispatch. Firstly, it's checked if any method, with the monitoring annotation specific to the interceptor, on the target object was ever called (via the first map). If not, then a method call dispatcher is created. On the method call dispatcher, it's checked if the method in the question on the current instance was ever called (the second map). If not, then create a method wrapper. When such method call wrapper is instantiating, it has access to the calling object, annotation values, and to the original method handler, so, it can customize the final behavior of the intercepted method based on those values. If such method call wrapper already exists, it's called. Within its implementation, usually some measuring logic wrapping the original method call is done. See this flow in the following Figure 5.2:

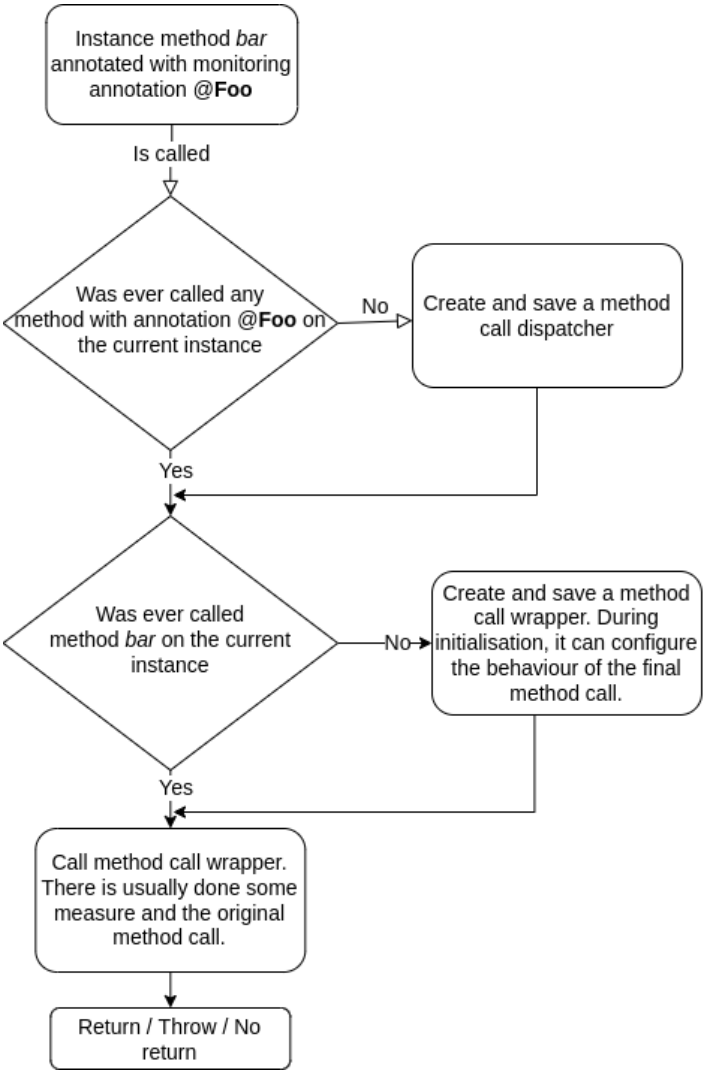

Figure 5.2: Flow of an annotated method call

When the interceptor for some type is created, then it's also registered into the gatherers registry. Interactions between thus dynamically created gatherers and the gatherers registry are shown in Figure 4.3. Metrics collecting is done via iterating over both maps while method call wrappers are also gatherers. The first map maps objects (of the type for which this interceptor was created) to the method call dispatchers. Methods call dispatchers map method name to method call wrappers. From this follows the need for the use of concurrent implementations for maps, as method call wrappers are created dynamically on the first method usage.

To implement other monitoring annotation, e.g. *TrackSomeAction* annotation listed in Listing 5.1, the installation class has to be provided, as it is shown in Listing 5.2. The interceptor class for TrackSomeAction is listed in Listing 5.3.

```java
/**
 * Monitoring method annotation @TrackSomeAction
 */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TrackSomeAction {
      ... // some annotation fields
}
```

Listing 5.1: TrackSomeAction method annotation

```java
/**
 * Installation for the method annotation @TrackSomeAction
 */
@Service
public class TrackSomeActionInstallation extends
    AnnotationInstallationBase<TrackSomeAction> {

  /**
     @param registryProvider - gatherers registry provides gatherers
          registry where the interceptor will be registered after its
          creation

     @param instrumentation - instrumentation to install on this
         installation
   */
  public TrackSomeActionInstallation(GatherersRegistryProvider
      registryProvider, Instrumentation instrumentation) {
    super(TrackSomeActionInterceptor::new, TrackSomeAction.class,
        registryProvider.getRegistry(), instrumentation);
  }

}
```

Listing 5.2: Installation class for @TrackSomeAction

```java
/*
 * Interceptor for the method annotation @TrackSomeAction
 */
public class TrackSomeActionInterceptor extends
    InterceptorForAnnotationBase {

  /**
   * @param originalTypeName name of type on which annotation was
       found
   */
  public TrackSomeActionInterceptor(String originalTypeName) {
      super(originalTypeName);
  }

  @Override
  public MethodCallWrapper createMethodCallWrapper(Object instance,
      Method originalMethod) {
    return new CallWrapper(instance, originalMethod, typeName);
  }
```

```
  private static class CallWrapper extends MethodCallWrapperBase<
      TrackSomeAction> implements MethodCallWrapper {

    /**
    * @param instance to be intercepted
    * @param method   method on the instance to be intercepted
    * @param typeName type name of instance class
    */
    public CallWrapper(Object instance, Method method, String
        typeName) {
      super(TrackSomeAction.class, instance, method, typeName);
      // Here we can configure the behavior of the `call` method
          based on the value from:
      // 1. the Method instance (java.lang.reflect) of the original
          method
      // 2. the instance on which the method is intercepted
      // 3. annotation values accessed via
      //       this.annotation.$SOME_ANNOTATION_FIELD
      ...
    }

    /**
    * @param callable   callable object of the original method
    * @param args       argument of the original method
    */
    @Override
    public Object call(Callable<?> callable, Object[] args) throws
        Exception {
      // this is called instead of the original method
      ... // do some measure, we have also access to the method call
          arguments
      Object result = callable.call(); // call the original method
      ... // do some measure
      return result;
    }

    /**
    * @return  metrics related to the one method, usually
        accumulated during method calls
    */
    @Override
    public Metrics getMetrics() {
      // fill Metrics object with up-to-date metrics, usually
          creating the new instance of Metrics class
      ...
      return metrics;
    }
  }

}
```

Listing 5.3: Interceptor for @TrackSomeAction

The overview of the important interfaces and classes used to implement monitoring annotations is in Figure 5.3:

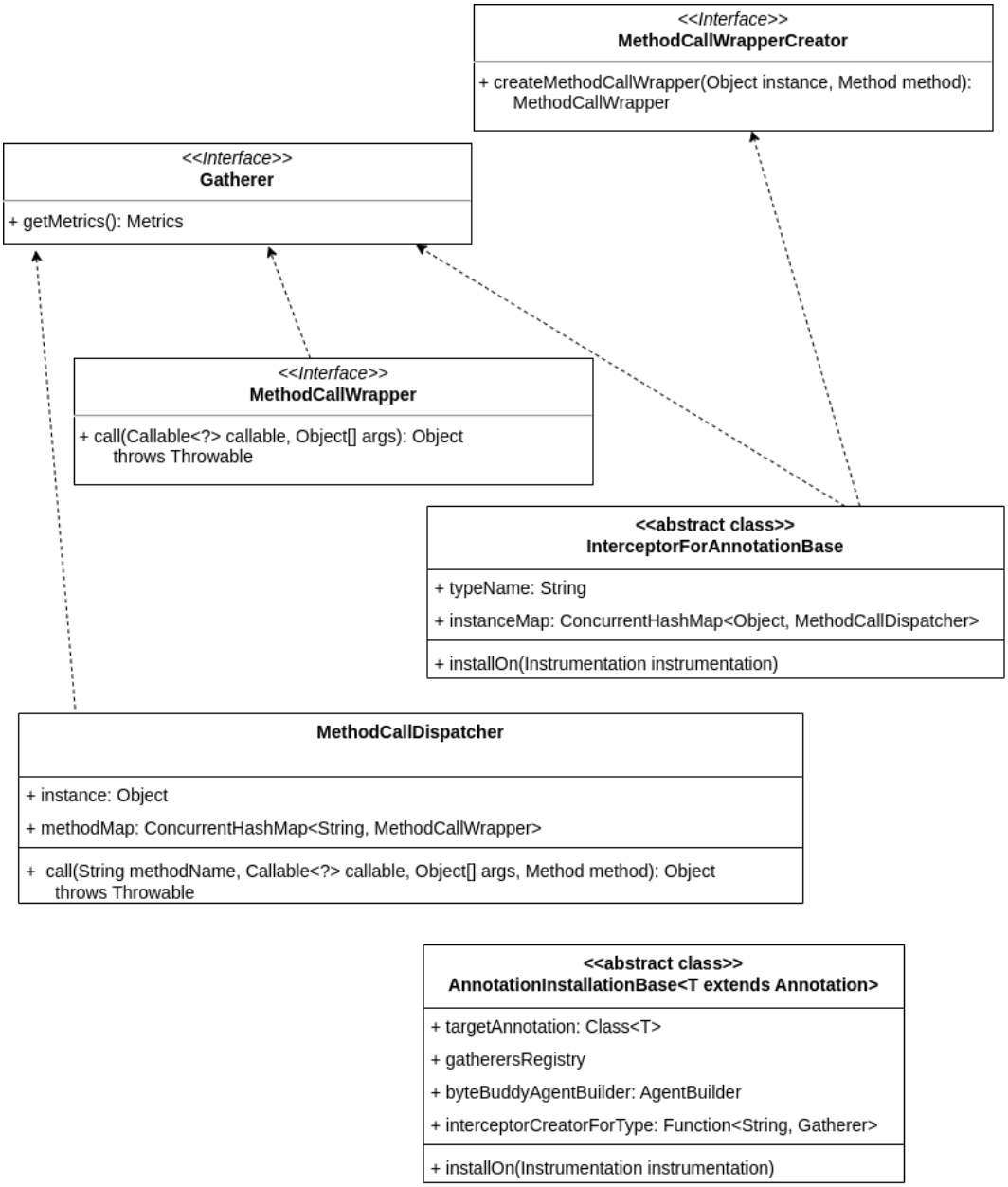

Figure 5.3: Class hierarchy for monitoring annotations implementation

5.2 Monitoring Java Agent Bootstrapping

When we were implementing the monitoring Java agent, we encountered some class loading problems. When we attached the monitoring Java agent to some Spring application that uses the same logging framework Log4j as our monitoring agent, the logging framework wasn't correctly initialized. After the investigation, we found out, that Java agents are loaded by the System class loader. This causes that the Java agent could use classes and resources from the target application and vice versa. This could cause classes clashes and bring unexpected behavior, especially for frameworks that do complex logic on their initialization. This was also our case. To resolve this issue, we tried to use Apache Maven Shade Plugin to shade (rename/realocate) the packages of the used dependencies. At the first

sight, it looked like it works, but it wasn't a clean solution. The annotation support stopped working.

The problem was, that monitoring annotations and auxiliary monitoring types used in the target application were loaded by a different class loader than monitoring annotations used in the code which determines if some class transformation should be taken. The expected actions weren't taken because the same class definition loaded by different class loaders is seen as two distinct classes by the JVM. We use type equality tests on annotations to determine if the interceptor for that annotation should be instantiated.

Class loaders in Java are responsible for dynamic loading Java classes to the JVM during runtime. Those classes are loaded on demand. Class loaders in Java form a class loader hierarchy. Class loaders use the delegation model, where on request to find a resource or class, a class loader instance will delegate the search of not loaded entity to the parent class load loader. Only if the parent delegation search was unsuccessful, it tries to search an entity by itself. Standard Java class loader hierarchy consists of the root Bootstrap class loader. Its child is the Extension class loader and on the last level is the System class loader (see Figure 5.4).

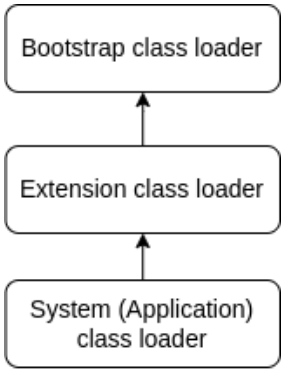

Figure 5.4: Standard Java class loader hierarchy

To resolve our equality and visibility problems, we implemented a very thin Java agent, which contains two jar libraries. One is the real monitoring agent implementation and the second one is the library jar file. The library jar file contains interceptors and monitoring types (annotations, interfaces, classes) that can be used in the user code, via monitoring API, and also in the monitoring agent implementation. The monitoring agent is then responsible only to unpack those jars and append them to the appropriate class loaders. Over the monitoring agent implementation's jar file we create a new *URLClassLoader*, a class loader capable of loading classes and resources from the jar files, and append it under the parent of the System class loader. The library jar file is appended to the Bootstrap class loader search path, via the method on the instrumentation object (from java.lang.instrument). This is illustrated in Figure 5.5.

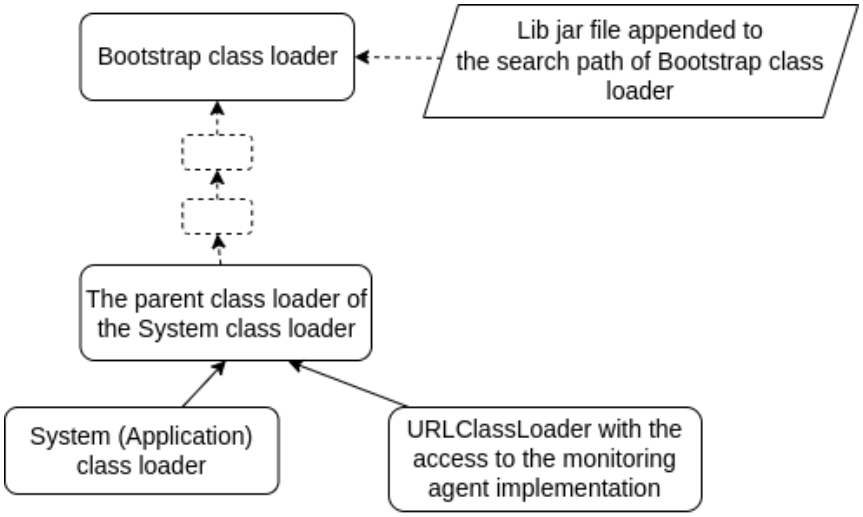

Figure 5.5: Class loader hierarchy while using monitoring agent

With this approach, both the target application and monitoring agent use the same common classes and interfaces loaded by the Bootstrap loader and they can interact via them. Using own class loader for internal implementation and dependencies for the monitoring agent implementation we avoid classes clashes, as those classes and resources are not visible to each other across our class loader and the target application class loader (usually the System class loader or a deeper class loader, one of its descendants).

5.3 MANTA Admin GUI Integrations

On the MANTA Admin GUI side, we implemented JMS listeners to consume metrics messages by *Consumer*. We implemented one consumer, which uses *Publisher* implementation to publish metrics data into the file system in a form of CSV files, as described in Section 4.5. The overview of classes and interfaces of used in implementation is in Figure 5.6.

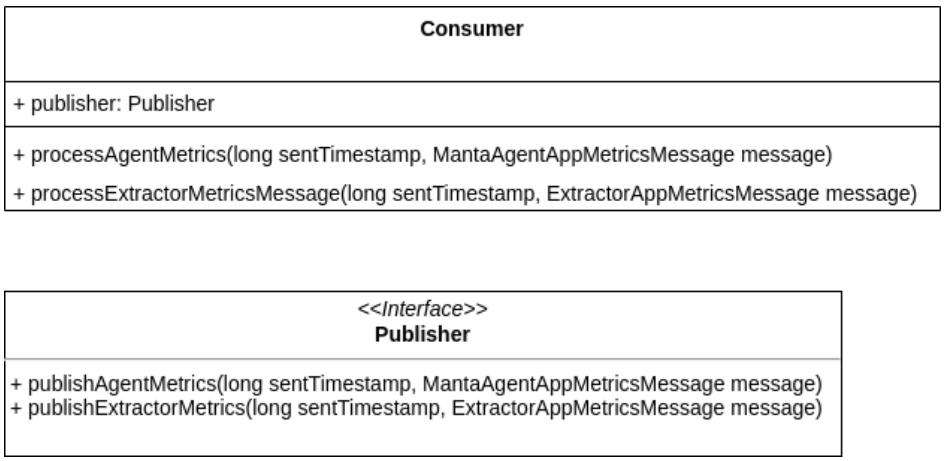

Figure 5.6: Classes and interface used by metrics messages listeners

CSV file publisher deserializes metrics messages by a deserializer determined by the JMS header value. We currently use JSON format as a transport format.

The file publisher also uses metadata from the message headers, which determines the final paths of the metrics files. The overview of metrics messages and their headers can be found in Section 4.3. Metrics data are by default stored in the directory `${installDir}/serviceutility/webapps/manta-admin-gui/WEB-INF/data/agent-metrics`. In the following Listing 5.4, we can see where the metrics file could be stored in the file system. We also implemented a mechanism to limit the maximum size of metrics files. If the file size achieves a defined limit, the file is rolled out to the file with the suffix *old*.

```
agent-metrics
├─ Agent[5a670501-72bf-4576-9d36-8c9dd94f405e] # agent ID
│  ├─ agent
│  │  └─ metrics.csv
│  └─ workflow
│     ├─ 3 # workflow execution ID
│     │  ├─ postgresqlDictionaryMappingScenario # scenario name
│     │  │  ├─ Postgresql-DEV-3 # connection name
│     │  │  │  └─ metrics.csv
│     │  │  └─ Postgresql-DEV-1
│     │  │     └─ metrics.csv
│     │  └─ postgresqlExtractorScenario
│     │     └─ Postgresql-DEV1
│     │        ├─ metrics.1644505114780.old
│     │        ├─ metrics.1644505314780.old
│     │        └─ metrics.csv
│     └─ 5
│        ├─ postgresqlDictionaryMappingScenario # scenario name
│        │  ├─ Postgresql-DEV-3
│        │  │  └─ metrics.csv
│        │  └─ Postgresql-DEV-1
│        │     └─ metrics.csv
│        └─ postgresqlExtractorScenario
│           └─ Postgresql-DEV1
│              └─ metrics.csv
└─ Agent[2c61bef0-6c4c-4e15-b4a0-d1417527376f]
   ├─ agent
   │  └─ metrics.csv
   └─ workflow
      └─ 4
         ├─ postgresqlDictionaryMappingScenario
         │  ├─ Postgresql-DEV-3
         │  │  └─ metrics.csv
         │  └─ Postgresql-DEV-1
         │     └─ metrics.csv
         └─ postgresqlExtractorScenario
            └─ Postgresql-DEV1
               └─ metrics.csv
```

Listing 5.4: Metrics files on the file system

We didn't implement any other publishers and integrations. The most critical integration, feeding metrics data into Prometheus has to be more analyzed, as it would probably cause bigger changes to the current MANTA Flow Platform deployment process. This is still an open question as MANTA currently is experiencing huge architectural changes. For now, we just use a script that can fill metrics data from the metrics files into the Prometheus database.

6. Evaluation

Within this chapter, we shortly summarize the limitations of the current implementation and how to use monitoring support for MANTA Flow Agent from the user and develop perspective.

6.1 Usage

To use annotation-based monitoring API from the Section 4.4.1, the developer has to import them from the module *manta-flow-agent-monitoring-agent-annotations* (see Attachment A.1). They can be used to track some aspects of an instance method. If the user wants to implement custom metrics providers (see Section 4.4.2), the module *manta-flow-agent-monitoring-agent-client* (see Attachment A.1) has to be used, as it contains the class *Metrics* which is coupled with the custom metrics providers usage. In both cases, to activate the monitoring method annotations functionality, the class has to be annotated with the marker annotation *@Monitor*. More details about the process of implementing new annotations and internals can be found in Section 5.1.3.

MANTA Flow Agent (which is not shipped as a thesis attachment) currently contains two types of startup scripts. One can be used to start MANTA Flow Agent with enabled monitoring supported for the main MANTA Flow Agent process and the second one without such support. On the MANTA Admin GUI side, the user can select if the extraction processes should be monitored or not. That global option per technology can be found in the *Configuration* tab, under the *CLI* options. There the user can choose for which technologies the MANTA Flow Agent can be used. Now, only four technologies (Postgresql, Oracle, Qlik Sense, MSSQL) can run the extraction phase on MANTA Flow Agent. The advanced configuration has to be edited to enable this, as it is shown in Figure 6.1. In this figure, we can see that we set *Collect runtime data* option to true and *Sampling interval* to 3 (seconds) for MSSQL global configuration. Alternatively, this can be set per connection, similar to the previous case, while defining a new connection. More information can be found in the MANTA Admin GUI user manual.

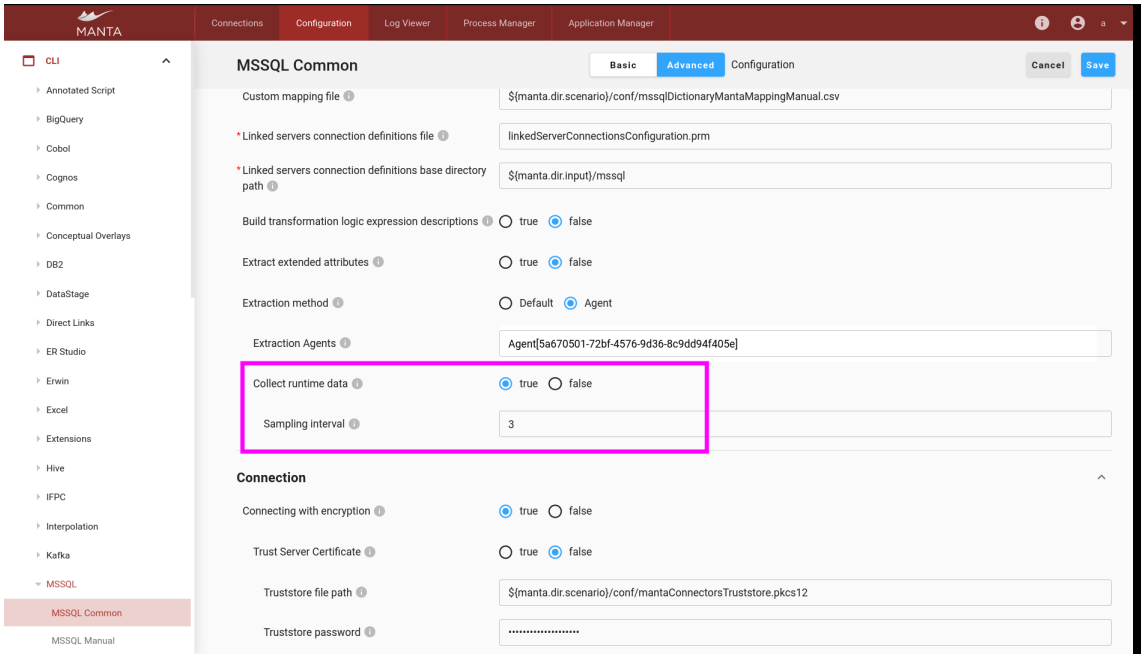

Figure 6.1: Configuring MANTA Admin GUI

Currently, only CSV File Writer for metrics data is implemented. Those metrics CSV files will appear under the `${installDir}/serviceutility/webapps/manta-admin-gui/WEB-INF/data/agent-metrics`. The created directory structure is described in Listing 5.4 and an example of metrics data in a CSV file can be found in Listing 4.7 or in *sample-metrics.csv* file from Attachments A.1. To export values from CSV files to the Prometheus database, one can use provided script *convert-csv-to-prometheus.py* (see Attachment A.1). This process is more described at the head of the script.

One can use the Prometheus default visualization tool to explore the gathered data. This is shown in Figure 6.2, alternatively, Grafana could be used, see Figure 3.8. Other data integrations (see Section 4.5 and Section 3.4.5) are not configured/implemented as this is still an open question and out of the scope of our Proof Of Concept implementation.

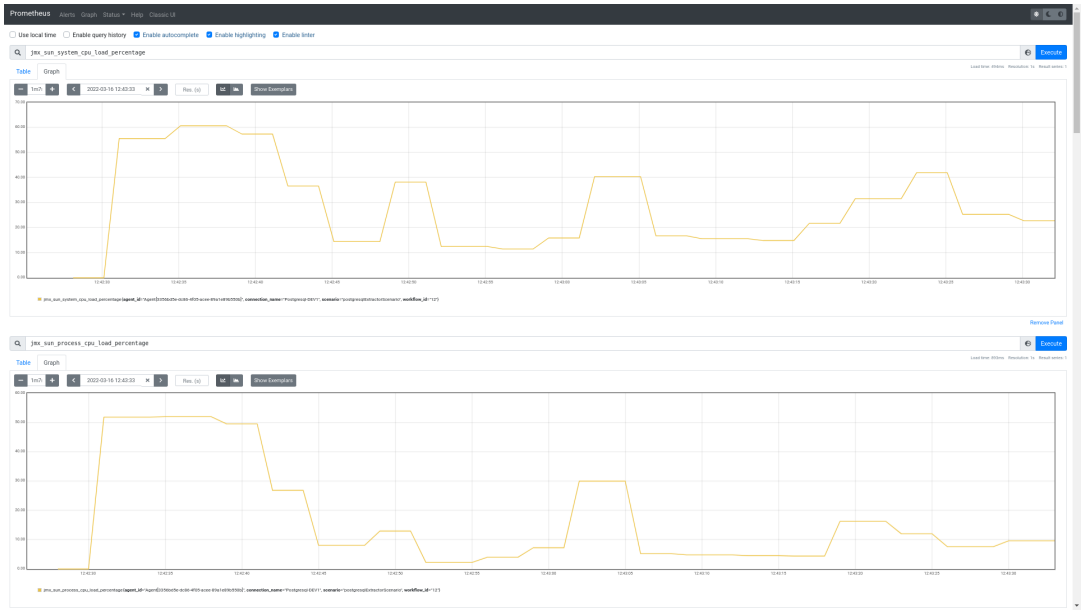

Figure 6.2: Visualized metrics data in Prometheus visualization tool

6.2 Testing and Limitations

As a result of this thesis, we provide an implementation of a Java agent, a monitoring agent. For this implementation, we implemented integrations tests to test the expected functionality. In those tests, MANTA Admin GUI, monitoring agent and MANTA Flow Agent are participating.

The current implementation has some limitations, especially for monitoring API. One of them is that method annotation can be used only on method instances (no static method support). Also, it is not possible to use them on the interfaces. Another limitation is that at most one monitoring annotation can be attached to the method, so they cannot be combined.

We also encountered some Byte Buddy limits which forced us to use the concurrent map for intercepted method dispatching (more details in Section 5.1.3). This intuitively limits us to using monitoring annotation on not hot functions. This is not so big problem, as our methods of interests usually deal with network. However, no automatic performance overhead measurements weren't implemented yet, only manual ones (running full extraction with enabled/disabled monitoring support) which showed no significant slowdowns (extractions are IO bounded).

More attention should be given to the main gathering loop and we should implement more defensive mechanisms to forbid the possibility of gathering blocking. See Section 4.2 where we discuss the roots and the possible solutions.

7. Conclusion

In this thesis, we presented the basic traits of MANTA Flow Agent and its role in MANTA Flow Platform. The most important aspects, such as interactions with other components from MANTA Flow Platform, concretely MANTA Admin GUI, and deployment plan were described. We discussed about the motivation and the importance of bringing monitoring support for MANTA Flow Agent.

To design and implement such support, we collected functional and non-functional requirements. We identified many functionalities which could the future monitoring support for MANTA Flow Agent provide us. The most notable are: the support for the collection of different metrics kinds (JVM metrics, system metrics, current process metrics, user-defined metrics), metrics inspection and anomalies detection.

Then we presented the analysis of different monitoring approaches and monitoring technologies suitable for our use case. Some of them were more general and some of them were more specialized for Java program monitoring, since MANTA Flow Agent is a Java application. None of the existing solutions could be easily used as-is. This was usually caused by our non-functional requirements. Some workarounds, which could be used, exist, but they usually require more non-standard modifications to the existing solution and to the recommended usage. Licensing could cause other problems too.

Finally, we decided to design and implement a custom solution based on the Java agent, which is very often used for monitoring purposes. We suggested using time series database for storing metrics. Prometheus was selected as a time series database. There was also discussion about why this one is preferable (see Section 3.4.3). One of its advantages is the existence of integrations, which could be used to fulfill some of our functional requirements, so we wouldn't have to provide custom implementations to fulfill them.

The current prototype implementation consists of the monitoring Java agent, which is capable to gather most of the metrics mentioned in the requirements. To collect user metrics data, we decided to implement annotation-based API as a form of non-intrusive API (if the monitoring is disabled, then the original code/behavior won't change). Internally it uses byte code instrumentation. This was the root of class loader related problems, which we struggled with. It resulted in a special bootstrap process implementation.

Metrics data gathered by the Java agent are written on the MANTA Admin GUI side, into CSV files only. Other integrations are not fully implemented. To import data into Prometheus, a manual step is required. Other Prometheus integrations have to be manually configured. The full integration requires more analysis, especially for deployment phases.

There are many directions in which the current implementation could be improved. We should investigate more the Prometheus data model and the current metrics name/labels to utilize and not misuse labeling functionality. The wrong usage could cause higher DB pressure. Annotation API should be revised to be more pleasant to use, e.g. the most used cases could have their own annotations.

Bibliography

[1] Oracle. Java™ Management Extensions (JMX™) Specification, version 1.4. https://docs.oracle.com/javase/8/docs/technotes/guides/jmx/JMX_1_4_specification.pdf.

[2] Spring Boot Actuator Web API Documentation. https://docs.spring.io/spring-boot/docs/2.6.7/actuator-api/htmlsingle.

[3] Oracle. Java Native Interface 6.0 Specification. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html.

[4] Oracle. JVM™ Tool Interface, Version 1.2. https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html.

[5] Apache Commons BCEL™ Documentation. https://commons.apache.org/proper/commons-bcel/.

[6] ASM - Java bytecode manipulation and analysis framework. https://asm.ow2.io/documentation.html.

[7] BTrace - A safe, dynamic tracing tool for the Java platform. https://github.com/btraceio/btrace/wiki.

[8] Byteman Programmer's Guide. https://downloads.jboss.org/byteman/4.0.18/byteman-programmers-guide.html.

[9] Javassist - Java bytecode engineering toolkit. https://www.javassist.org/.

[10] Byte Buddy - a code generation and manipulation library. https://bytebuddy.net/#/tutorial.

[11] JNIF - Java Native Instrumentation Framework. http://sape.inf.usi.ch/jnif.

[12] Package java.lang.instrument. https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html.

[13] OpenTelemetry Concepts. https://opentelemetry.io/docs/concepts/.

[14] Micrometer facade. https://micrometer.io/docs.

[15] Java Metrics Library. https://metrics.dropwizard.io/4.2.0/manual/index.html.

[16] Kiran Oliver. Prometheus and the Debate Over 'Push' Versus 'Pull' Monitoring. https://thenewstack.io/exploring-prometheus-use-cases-brian-brazil/.

[17] Giedrius Statkevičius. Push Vs. Pull In Monitoring Systems. https://giedrius.blog/2019/05/11/push-vs-pull-in-monitoring-systems/.

[18] Push vs Pull. https://blog.sflow.com/2012/08/push-vs-pull.html.

[19] Steve Mushero. Push vs. Pull Monitoring Configs. https://steve-mushero.medium.com/push-vs-pull-configs-for-monitoring-c541eaf9e927.

[20] Kovid Rathee. The Case for Using Timeseries Databases. https://towardsdatascience.com/the-case-for-using-timeseries-databases-c060a8afe727.

[21] How does OpenTSDB work? http://opentsdb.net/overview.html.

[22] Introduction to QuestDB. https://questdb.io/docs/introduction.

[23] TimescaleDB Overview. https://docs.timescale.com/timescaledb/latest/overview/.

[24] What Graphite is and is not. https://graphite.readthedocs.io/en/1.1.8/overview.html.

[25] Get started with InfluxDB OSS 2.2. https://docs.influxdata.com/influxdb/v2.2/.

[26] What is Prometheus? https://prometheus.io/docs/introduction/overview/.

[27] OLAP for Multidimensional Analysis. https://olap.com/olap-definition/.

[28] Introduction to Grafana. https://grafana.com/docs/grafana/v8.4/introduction/.

List of Figures

Listings

A. Attachments

A.1 Content of the attached ZIP file

- **README.md** contains information about the content of the attached ZIP file.
- **sample-metrics.csv**: a sample of metrics data in CSV format.
- **src**: directory containing the following files and directories
  - **monitoring-agent**: module implementing Java monitoring agent.
  - **manta-admin-gui-agent-metrics-logic**: module for MANTA Admin GUI to integrate monitoring support.
  - **manta-flow-agent-integration-tests**: module containing integration tests.
  - **convert-csv-to-prometheus.py**: script which can be used to convert and feed metrics data from CSV files to Prometheus time series database.