Project Title	Artificial Intelligence in Secure PRIvacy-preserving computing coNTinuum
Project Acronym	AI-SPRINT
Project Number	101016577
Type of project	RIA - Research and Innovation action
Topics	ICT-40-2020 - Cloud Computing: towards a smart cloud computing continuum (RIA)
Starting date of Project	01 January 2021
Duration of the project	36 months
Website	www.ai-sprint-project.eu/

D3.2 - First release and evaluation of the monitoring system

Work Package	WP3 | Requirements & Architecture Definition
Task	T3.3| Application, Infrastructure and Edge Monitoring
Lead author	Radosław Ostrzycki (7BULLS),
Contributors	Grzegorz Timoszuk(7BULLS),
Peer reviewers	Danilo Ardagna (POLIMI), Amanda Calatrava (UPV)
Version	V1.0
Due Date	31/12/2021
Submission Date	22/12/2021

Dissemination Level

X	PU: Public
	CO: Confidential, only for members of the consortium (including the Commission)
	EU-RES. Classified Information: RESTREINT UE (Commission Decision 2005/444/EC)
	EU-CON. Classified Information: CONFIDENTIEL UE (Commission Decision 2005/444/EC)
	EU-SEC. Classified Information: SECRET UE (Commission Decision 2005/444/EC)

Versioning History

Revision	Date	Editors	Comments
0.1	22.11.2021	Radosław Ostrzycki	1st initial version of 1-3 sections
0.2	26.11.2021	Radosław Ostrzycki	1st initial version of 4-8 sections
0.3	27.11.2021	Grzegorz Timoszuk	Internal revision and filling gaps
0.4	27.11.2021	Radosław Ostrzycki	Clearing comments and extending document
0.5	9.12.2021	Danilo Ardagna	Document review
0.6	14.12.2021	Amanda Calatrava	Document review
0.7	10.12.2021	Radosław Ostrzycki	Corrected document structure after review (work in progress). Sections 2-8
0.8	14.12.201	Radosław Ostrzycki	Changing document structure after review (work in progress). Sections 2-8.
0.9	15.12.2021	Radosław Ostrzycki	Changing document structure after review (work in progress). Sections 2-8.
0.10	19.12.2021	Grzegorz Timoszuk	Update of Executive summary and introduction
0.11	20.12.2021	Danilo Ardagna	2nd document review
0.12	20.12.2021	Amanda Calatrava	2nd document review
0.13	21.12.2021	Grzegorz Timoszuk	Minor comments review
0.14	21.12.2021	Radosław Ostrzycki	Comments review, update document structure.
0.15	22.12.2021	Danilo Ardagna	3rd document review
0.16	22.12.2021	Radosław Ostrzycki	Comments review.
0.17	22.12.2021	Amanda Calatrava	3rd document review
1.0	22.12.2021	A Radosław Ostrzycki	Final edited version

Glossary of terms

Item	Description
OS	Operating System
K8s	Kubernetes
CEP	Complex Event Processing
HA	High Availability
IM	Infrastructure Manager
PoC	Proof of concept

Keywords

Monitoring; Artificial Intelligence; Edge Computing; Computing Continuum; Runtime management.

Disclaimer

Executive Summary

This document, developed by the AI-SPRINT project, describes the first release and evaluation of the monitoring subsystem. The final release and evaluation of the monitoring subsystem is due at M24.

The document describes the components involved in monitoring subsystems, as well as the results of the preliminary evaluation. The technological choices are taken considering the requirements elicited in *AI-SPRINT Deliverable D1.2 Requirements Analysis*. The document first introduces the functional and non-functional requirements of the monitoring subsystem. Next the focus is on an analysis and evaluation of the available open source tools. Finally, the document describes the innovative architecture that will be developed within the AI-SPRINT project which allows monitoring data collection also in case of edge/cloud disconnection and logs collection. The Proof of Concepts (PoCs) developed so far, the tests conducted and a roadmap for the development activities are also described.

It is worth mentioning that the architecture proposed in this document passed preliminary evaluation. The PoCs we implemented cover the most crucial parts of the implementation and tests in laboratory environments have been conducted. The results indicate that the monitoring subsystem will pass the requirements towards scalability and performance with no issues. The most important goals for the next twelve months are the implementation of means to easily define custom metrics, the support for hierarchical deployment and the automation of monitoring subsystem deployments and configuration.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Scope of the Document

The aim of the AI-SPRINT "Artificial intelligence in Secure PRIvacy-preserving computing coNTinuum" project is to develop a platform composed of design and deployment tools that will support rapid development of AI applications. AI-SPRINT will provide a cloud agnostic solution and it will optimize integration with AI-based edge sensor devices.

Providing a monitoring solution for AI-SPRINT will allow users to analyse various parameters describing the system state over time. It will also allow the automation of various tasks which must be performed when specific system working conditions are met. Also it is very important to provide the possibility of analysing data which comes from edge devices. It must be taken into account that some parts of the AI-SPRINT platform may work in separation (offline) for long periods of time. This will apply especially to edge devices which may be disconnected from central services.

Data collected by AI-SPRINT monitoring solution will contain system and custom (i.e., application-level) metrics as well as logs from AI-SPRINT-based applications. To support developers, AI-SPRINT will provide a Python library which will expose a high-level API which will allow seamless integration between Python applications and AI-SPRINT monitoring facilities.

This document will cover an in depth analysis of the state of monitoring at the computing continuum. The evaluation was done taking into account the following main principles:

1. Ability to work with edge devices (limited resources vs. typical servers/VM/containers).
2. Ability to handle disconnection (edge devices may lose/work in the conditions with no coverage).
3. Ability to be customised towards different requirements of different users, especially taking into account custom intervals, emiliting alarms, custom metrics, collecting custom data.
4. Ability to be adapted toward collecting application logs.

A new software stack containing InfluxDB, Telegraf, ElasticSearch and Grafana has been proposed. The mentioned tools will be put in a framework that will address the requirements mentioned above. What is more, for infrastructure management, AI-SPRINT developers also need to define, calculate and store custom metrics (for example: application level, container level, etc). AI-SPRINT will provide for this reason a high level library to easily define and store such metrics.

Additionally, the monitoring framework needs to fulfil requirements defined in the *AI-SPRINT Deliverable D1.2 Requirements Analysis*. There are six main areas of such requirements:

1. Data storage (data volume, data structure, querying, data retention)
2. Data acquisition (push and pull strategies, multiple levels: OS, cluster, pods)
3. Data processing (alerts, aggregation, hierarchies)
4. Security (data access, secure connections)
5. Deployment and execution environment (K8S support, automation)
6. Support for developers (covered by the library for custom metrics mentioned above).

The proposed architecture will address the requirements mentioned above. It will provide building hierarchical structures which will be able to support complex cluster architectures and multiple edge devices. By default, there will be multiple basic metrics predefined and the Monitoring Subsystem will have the ability to visualize data.

Finally a preliminary evaluation of the Proof of Concepts (PoC) developed so far will be presented. It will tackle multiple deployment scenarios and multiple usage scenarios. An important aspect of each monitoring solution is the impact on the cluster's performance and load (both CPU and memory) which will also be covered and discussed in detail. Performance tests were conducted in a laboratory environment to confirm that impact on the system is within expected boundaries.

1.2 Target audience

This document is intended for all the actors, project partners and external stakeholders involved in the AI-SPRINT project. It is intended for internal use, although it is publicly available. The target audience is the AI-SPRINT technical team including all partners.

1.3 Structure

The document is organised as follows: in Section 2 both functional and non-functional requirements for the monitoring system are summarised. Next, chapter 3 reports the technology evaluation we performed to select the tools and technologies that will be the AI-SPRINT monitoring framework building blocks. Section 4 summarises the envisioned AI-SPRINT monitoring architecture and section 5 covers the system deployment conducted so far. After that, section 6 contains the current evaluation that also covers performance testing. Finally, section 7 contains future plans and section 8 provides conclusions.

2. Monitoring subsystem

Monitoring is a necessary and important part of each advanced IT system. It allows system administrators to have an in-depth view of overall system state and its performance. Monitoring also provides the ability to track the root cause of issues and trigger automatic actions in case of anomalies. In general, it is assumed that such solutions should not interfere with other systems, work seamlessly in the background and require virtually no maintenance. They also should provide administrators (their main users) a predefined set of functionalities available out of the box and tools to provide advanced data analysis. Another important aspect is keeping the impact of the monitoring on the whole system performance on a minimal level.

This section will describe the AI-SPRINT Monitoring Subsystem. At first its main, generic responsibilities will be introduced. Then we will present all requirements, which were gathered during the project analysis phase. We decided to group them into a few categories. This analysis will help during the technology evaluation phase, which will be presented in the next section.

2.1 Introduction

The Monitoring Subsystem will be an independent module of AI-SPRINT. It will be responsible for gathering and processing time series data that are describing various parameters of other subsystems. In this document, these time-indexed parameters are called “metrics”. In general, a typical “metric” consists of:

- a timestamp;
- a value: scalar or vector, mostly of numeric type, but sometimes may be a text (ex. status, state etc);
- a set of tags (labels) in form of pairs: (tag name, text tag value), which describe the meaning and the context of a metric.

Also the Monitoring Subsystem will allow to collect text logs generated by applications. Text logs will consist of:

- a timestamp;
- a text message (may be long);
- a set of tags (labels) in form of pairs: (tag name, text tag value), which describe the meaning and the context of a log record;

2.2 Requirements

During project analysis we gathered a broad set of requirements, which must be met by the new AI-SPRINT Monitoring Subsystem. Most of the requirements are summarized in AI-SPRINT Deliverable *D1.2 Requirements Analysis*. They can be grouped into six areas: data storage, acquisition and processing, security, deployment and support to the developers. All of them will be overviewed in the following subsections.

2.2.1 Data storage

Below we list the functional and non-functional requirements towards monitoring data storage:

- The Monitoring Subsystem must be able to store large amounts of data (hundreds of GBs).
- Every stored data record must contain a timestamp, a value and an optional set of tags (label and text pairs). The timestamp and the tags make the record unique.
- A record value may be a floating point number or a text (in case of log messages or text metrics).
- The Monitoring Subsystem should allow a user to fetch and query data using a set of filters.
- Data query language must support basic statistical functions (average, mean, sum).
- The data storage engine must provide administration tools (via UI and REST API).
- It may be necessary to implement data retention policies.

2.2.2 Data acquisition

Given a preliminary analysis of the AI-SPRINT use cases, a typical deployment will consist of about 10 nodes. Moreover, there will be no more than 100 monitored modules. In this scenario, below is the list of requirements toward monitoring data acquisition:

- The Monitoring Subsystem should be able to read (scrape, pull) metrics from monitored applications using HTTP endpoints.
- Monitored applications must be able to send (push) data to the Monitoring Subsystem.
- Data acquisition must be possible for short living tasks as well as long living processes (applications, systems.
- The Monitoring Subsystem has to be able to collect Kubernetes and cluster nodes statistics.
- Monitored applications may provide text logs to be preserved by the Monitoring Subsystem. These logs may be collected during application execution or from text files using dedicated tools which will allow importing these logs to the Monitoring Subsystem.

2.2.3 Data processing

Below are challenges set for the monitoring data processing:

- There must be the possibility to create alerts triggered when given conditions are met.
- When conditions are met for a metric, an alert should be triggered and a given HTTP endpoint must be called by the Monitoring Subsystem (this mechanism will be exploited by SPACE4AI-R and other runtime schedulers to trigger the application and system reconfiguration, see AI-SPRINT Deliverables *D1.3 Initial design of the architecture*).
- There must be a tool which will display collected metrics as graphs and allow users to define various data views.
- It must be possible to build hierarchies of data storage components and fetch or push collected metrics from one hierarchy level to another.
- It must be possible to display collected logs and search them by timestamp ranges. Users should be able to analyse logged data.

- When there are monitored components installed on separate clusters, which may work in isolation (edge devices), there should be the possibility to automatically send data gathered on isolated clusters to the central Monitoring Subsystem instance, after the network connection has been established.

2.2.4 Security

Below is the list of aspects tackling securing that should be taken into consideration during the implementation of the AI-SPRINT Monitoring Subsystem.

- Access to collected data must be secured.
- There must be a possibility to create and manage transparently the credentials used to connect to the Monitoring Subsystem.
- Every component of the Monitoring Subsystem must be able to run on the SCONE platform for security purposes.

2.2.5 Deployment and execution environments

Below is the list of requirements towards the deployment and execution environments:

- The Monitoring Subsystem must be able to be deployed on Kubernetes clusters.
- The deployment and configuration process must be able to be automated and adaptable to be used by Ansible (or similar) tools.
- Various elements of monitored infrastructure may be deployed on different, separate clusters. In such a case it should be possible to synchronize collected data between clusters.

2.2.6 Support for software developers

Below we report the list of requirements that will support developers creating solutions on the AI-SPRINT platform:

- Monitored applications must be able to send custom metrics to the Monitoring.
- The Monitoring Subsystem must provide a Python client library which will allow the application to send custom metrics and define alerting rules.

2.2.7 Support for application logs collections

Below is the list of requirements towards the gathering and browsing logs:

- Log files containing applications (jobs) text messages will be provided by external AI-SPRINT modules and then have to be loaded into the Monitoring Subsystem.
- Log files will be processed offline by a special “log processing module”.
- Log files must contain consistent date time information in each log record. Log record format (including fields) will be described later.
- Logs should be preserved in a database to be able to be searched and browsed using a web UI. Logs should be able to be correlated with other metrics.

3. Technology evaluation

Having collected the requirements of the project, it is possible to design the modules and their responsibilities that will build the AI-SPRINT Monitoring Subsystem. For that, we will start analysing the possibility of using existing software which provides monitoring capabilities. There is no option of using external, commercial monitoring service, because the expected solution should be able to work in controlled, isolated environments, and be extendable according to our needs. Furthermore, AI-SPRINT aims at developing open source solutions. To the best of our knowledge, after performing an extensive research and market analysis, there are no commercial tools that fulfil the given requirements. This is why we decided to focus on open-source software.

After the analysis of the AI-SPRINT requirements, we figured out that the Monitoring Subsystem will need:

- data gathering modules;
- a time series database;
- a data processing engine (probably closely connected with the database);
- a configurable, graphical web UI (for data presentation);
- a set of Python libraries providing access to data gathering modules, data processing engine and the database.

In general, data gathering modules offer two data collecting approaches: pull and push. The first one assumes that the monitored application should provide current metrics (statistics) on a given place (usually as a web endpoint) and then the data gathering module will pull (scrape) it periodically. In the second variant, the monitored application is responsible for providing metrics data to the data gathering service by sending it to a given destination (usually a web endpoint).

During the analysis and technology evaluation phase we reviewed a wide range of available tools, which may have been used as parts of the developed solution. Description of these tools and evaluation notes will be presented in the following subsections. First, we will describe tools which did not fulfil our requirements. Then we will describe components, which will be used as building blocks of the Monitoring Subsystem along with evaluation results.

3.1 Rejected tools

After a careful analysis some of the tools initially planned for the development of the Monitoring Subsystem had to be abandoned. The main reasons behind such decisions were conclusions made after in depth analysis of the requirements performed at the beginning of the AI-SPRINT project. In this section, we review all of the tools and services considered, justifying for each one of them why they have been discarded.

3.1.1 Esper

Esper is a Java library which provides Complex Event Processing features. It consists of a compiler and runtime module, which can be embedded in applications executed by Java Virtual Machine. It also defines a dedicated programming language used to define event processing rules. These rules are then compiled by the Esper Compiler and can be run by the Esper Runtime. As it is a Java library, it is not a complete, "out of the box" data processing engine. It is a base component to build custom data processing software.

Esper was developed by EsperTech Inc which provides commercial support and applications based on Esper (closed source).

Esper language is focused on analysing the stream of events. It may be useful in the case of providing "alerting" capabilities to the Monitoring Subsystem, but it does not allow performing data queries. Esper capabilities do not fulfil needed requirements. Furthermore, based on Java, it is sometimes too heavy to run in devices with limited capabilities widely used in edge/IoT systems and it does not match with the main language that will be considered by AI-SPRINT and most widely adopted by AI developers and data scientists, i.e., Python.

Pros:

- customizable CEP solution.

Cons:

- only a programming library;
- covers a small set of required features;
- big footprint due to Java language;
- may be hard to run on edge devices.

A summary of the evaluation process is presented in Table 3.1:

webpage	https://www.espertech.com/esper/
sources:	https://github.com/espertechinc/esper
license	GPL v2
module type	Java library, data processing engine
evaluation grade	covers a small range of requirements

Table 3.1 Esper evaluation summary

3.1.2 Melodic Monitoring EMS

The Melodic system, we initially planned to rely on at the proposal stage, allows the deployment and management of applications deployed in multi cloud environments. It contains a monitoring module which is responsible for analysing system metrics and events generated by clusters.

We analysed documentation and source code of the Melodic Monitoring EMS module. Unfortunately this module is tightly coupled with Melodic and cannot be easily run as a standalone application and would introduce a very significant integration effort with other AI-SPRINT components (mainly the Infrastructure Manager (IM), and SPACE4AI-R). Only its complex event processing capabilities potentially may be useful in AI-SPRINT's Monitoring Subsystem. That's why we decided not to use this product.

Pros:

- provides Complex Event Processing engine.

Cons:

- tightly coupled with the Melodic project, cannot be easily run as a standalone software;
- provides a small set of needed functionalities;
- does not support TOSCA;
- focused on Melodic multi cloud approach;
- no easy integration with K8S.

A summary of the evaluation process is presented in Table 3.2:

webpage	https://melodic.cloud
sources:	https://gitlab.ow2.org/melodic/melodic-upperware/-/tree/rc3.1/event-management
license	Mozilla Public License v2
module type	CEP engine
evaluation grade	provides a too small set of the needed functionalities

Table 3.2 Melodic Monitoring EMS evaluation summary

3.1.3 Prometheus

Prometheus is a monitoring server which is focused on gathering metrics from applications and system resources. It also contains a storage engine which allows querying collected data using a SQL-like query language. It is very important to understand that it is not a general purpose time series database, but rather a system which collects statistics (which may be aggregated) and is able only to pull (scrape) metrics provided by other applications.

Prometheus also comes with a set of tools which enhance its capabilities. One of them is a "Pushgateway" which allows short-living applications (jobs, tasks) to push metrics. It is de facto a "metrics cache", because it provides collected data to be pulled (scraped) from the Prometheus instance.

Another relevant tool is an Alertmanager which is responsible for aggregating alerts generated by prometheus and sending them to appropriate consumers. Alerts' destinations are defined in a text YAML file and allow to publish alerts on generic web endpoints and on a few popular services (ex. Slack, AWS SNS).

Prometheus as an all-in-one tool seemed to be a 1st and main choice as a base of AI-SPRINT's Monitoring Subsystem. It offers rich data query language, which is based on SQL. Queries can be sent via a well-defined REST API which makes Prometheus a universal analytics tool. However it appeared that it may be difficult to build required functionality using it as a data storage or a data acquisition module. Loading data from external sources and preserving its timestamps is difficult to achieve. This would make it difficult to construct hierarchical structures. Also it can only load data (pull) from monitored applications and does not provide the possibility to send metrics with timestamps by applications (push). This functionality is crucial, because AI-SPRINT will have to provide monitoring for short-living tasks. As there were good alternatives to prometheus, we decided not to use it in the AI-SPRINT Monitoring System.

Pros:

- "all in one" tool focused on metrics gathering, storage and analysis;
- configuration stored in YAML files;
- input metrics format is a de facto standard;
- rich query language based on SQL;
- queries can be sent to the REST API;
- can transparently send and receive data from external data storage systems;
- millisecond timestamp resolution.

Cons:

- backfilling metrics data (importing older data) is difficult and can cause problems;
- can only pull data (scrape);
- timestamps assigned when a metric is read from an application;
- the management REST API is very poor, users cannot change configuration using REST API;
- alerting rules and routes can be defined only in files (no REST API).

A summary of the evaluation process is presented in Table 3.3:

webpage	https://prometheus.io/
sources:	https://github.com/prometheus/prometheus
license	Apache 2.0
module type	database, data gathering, processing engine
evaluation grade	does not fit AI-SPRINT requirements

Table 3.3 Prometheus evaluation summary

3.1.4 Graphite

Graphite is a monitoring tool which provides storage for time series data and a presentation web UI. By default it uses an internal data storage engine called Whisper, which provides one second precision for saved metrics. This is a fixed-size database, similar to RRD (round-robin-database), which saves space on disk by changing data resolution for older entries (lossy data storage). Queries also can be executed by using a REST API which provides simple query syntax with a broad collection of aggregation functions. Administrators can define new functions in the Python language. Graphite also provides a useful, graphical UI that allows users to browse and query collected data.

As an all-in-one monitoring tool, Graphite seemed a good choice at the first sight. Unfortunately, provided one second time precision is too small for AI-SPRINT. Provided lossy data storage in RRD is also not acceptable. That's why we decided no to use this tool as a building block in the Monitoring Subsystem.

Pros:

- all-in-one tool;
- written in Python, can be run on SCONE;
- simple query language with a broad, extendable collection of aggregation functions.

Cons:

- only second timestamp resolution;
- not a general-purpose time series database, lossy data storage (RRD).

A summary of the evaluation process is presented in Table 3.4:

webpage	https://graphiteapp.org/
sources:	https://github.com/graphite-project
license	Apache 2.0
module type	time series database and presentation tool
evaluation grade	lossy data storage is not acceptable

Table 3.4 Graphite evaluation summary

3.1.5 StatsD

StatsD is a network service for collecting and relaying collected statistics. It allows clients to send metrics (push metrics), which then are locally buffered and sent to an external data storage called "backend". There are at about 30 backends available, however only half of them may be considered as stable and reliable.

As another all-in-one tool, StatsD was considered as a good base for building the AI-SPRINT Monitoring Subsystem. It provides a broad range of backend (data storage) engines. It can also receive data from external applications (push method), which may be used by monitoring client library to push metrics from short-living tasks. Unfortunately provided adapters, for storing data in various databases, are in many cases not reliable. StatsD does not provide any analytics tools and cannot scrape (pull) prtrics from monitored applications. As there are other, better tools to store and gather metrics, we decided not to use it.

Pros:

- clients can send data to the StatsD server (push);
- many backends available.

Cons:

- cannot scrape (pull) metrics;
- many backends are not stable or reliable;
- written in JavaScript, needs Node.js to run.

A summary of the evaluation process is presented in Table 3.5:

webpage	https://github.com/statsd/statsd
sources:	https://github.com/statsd/statsd
license	MIT
module type	Metrics collection and buffering
evaluation grade	covers a small set of needed functionalities

Table 3.5 StatsD evaluation summary

3.2 Chosen tools

After a careful analysis, internal discussion and joint evaluation, a set of tools has been selected that will be used as building blocks for the AI-PRINT Monitoring Subsystem implementation. In this subsection we analyse them and justify why they have been chosen.

3.2.1 InfluxDB

InfluxDB is a time series NoSQL database focused on storing and analyzing collected metrics. It provides a specialized language (Flux), which allows performing advanced queries on the gathered data. There is also a built-in mechanism of asynchronous task execution, on top of which is built an alerting module. InfluxDB also contains a web UI, which can be used for database administration and data visualization purposes. It provides a rich REST API, so it can be easily integrated with external systems. All these features make InfluxDB a complete solution for data storage, visualization, processing, and alerting.

InfluxDB was developed by Influx Data, which also provides a commercial support and cloud based services.

During our evaluation we created a few proof-of-concepts and small projects to check the potential usefulness of this database. We compared features provided by this database towards storage engine and data processing requirements for the AI-SPRINT Monitoring Subsystem. InfluxDB fulfils most of them, so we decided that this database will be responsible for core functionalities in these areas.

Pros:

- full featured product;
- open source;
- database focused on processing time series data;
- supported, new versions are released very often (bug fixes and enhancements);
- can be easily deployed on K8s using the Helm tool (deployment can also be customized);
- written in Golang, can be deployed on SCONE;
- provides a REST API with a broad set of functions;
- web UI management interface;
- web UI data presentation interface;
- rich set of client libraries (Python, Java, etc);
- can fetch data from SQL databases;
- can fetch and send data to other InfluxDB instances (one can build a tree of InfluxDBs);
- contains advanced alerting mechanisms;
- Flux language contains a huge set of data processing functions;
- native integration with Telegraf, Prometheus and other data gathering tools;
- data retention policies can be defined;
- nanosecond timestamp granularity.

Cons:

- very limited and basic High Availability and clustering capabilities in open source version (InfluxDB OSS);
- async tasks are hard to maintain;
- non-standard Flux language focused on time series data processing (can be difficult to understand and learn).

A summary of the evaluation process is presented in Table 3.6:

webpage	https://docs.influxdata.com/influxdb/v2.0/
sources:	https://github.com/influxdata/influxdb
license	MIT License
module type	database, data processing engine, web UI
evaluation grade	it covers a huge set of our requirements;

Table 3.6 InfluxDB evaluation summary

3.2.2 Telegraf

Telegraf is an application dedicated to gathering metrics, local buffering and sending them to various destinations. This tool is very customizable and has over a hundred input and output plugins, which integrate it with many data sources. Telegraf is well-integrated with InfluxDB and can receive or scrape data from measured applications (push and pull metrics). Thanks to one of the input plugins, it can also substitute Prometheus (which is de facto standard as a metrics gathering tool).

Telegraf was developed by Influx Data in the Go language. It is supported and actively extended by its creators and community.

Telegraf capabilities were checked and we validated its capabilities by implementing a few proof-of-concept applications. We checked how it cooperates with InfluxDB and how to deploy it on K8s cluster. We tested typical K8s deployment as well as Helm charts. There are two charts which provide either a typical “daemon set” and an “operator” deployments. The first one is ideal for providing full K8s cluster deployment while the second one is typically used for providing close integration with monitored applications as a monitoring “sidecar”. Unfortunately, the initialization time of such sidecar can be longer than a monitored application, which makes it unusable for short-living jobs. We observed that in such situations some metrics sent before Telegraf finished its initialization were lost. In such cases, Telegraf should be deployed as a typical service.

As Telegraf is a very good, proven, data acquisition tool, we decided to use it as a main solution in this area.

Pros:

- provides a pull and push data gathering model;
- open source;
- can be easily deployed on K8s cluster and a Helm chart is available;
- can be deployed on K8s as a service, daemon set and “operator” (sidecar);
- Provides many input and output plugins, highly configurable;
- can be run by SCONE;
- accepts Prometheus input data format;
- well-integrated with InfluxDB;
- many client libraries for various programming languages (including Python), these libraries allow pushing custom metrics to Telegraf instances from applications.

Cons:

- problems with full K8s monitoring in latest version of Telegraf helm chart due to changed default security configuration;
- Telegraf “sidecar” initalization time can be too long in some scenarios.

A summary of the evaluation process is presented in Table 3.7:

webpage	https://docs.influxdata.com/telegraf/v1.20/
sources:	https://github.com/influxdata/telegraf
license	MIT License
module type	data gathering
evaluation grade	very useful;

Table 3.7 Telegraf evaluation summary

3.2.3 ElasticSearch

ElasticSearch is a NoSQL database and a search engine which provides exceptional searching features for textual data. It is usually used as a supporting data search engine for custom applications or as a text logs and metrics storage engine. It is one of the elements creating a so-called ELK stack (ElasticSearch, Logstash, Kibana) providing widely-used, full log processing solutions for distributed systems.

After confirming that ElasticSearch will provide required log processing capabilities, we deployed it on the local test K8s cluster using official Helm’s charts. We successfully checked if it is possible to import text logs into ElasticSearch and then to display them in Grafana. We used a “typical” deployment configuration, which deploys this search engine as StatefulSet, but it is worth mentioning that there is a Helm chart which provides deployment as a K8s operator. In that case, administrators can use K8s custom resource definitions in order to define high-level definitions of running ElasticSearch clusters. There is one important aspect of using this tool in IT projects: its license model. It forces developers to publish source code of the whole system which provides ElasticSearch as SaaS to external systems. After analysing requirements, we think that this will not affect AI-SPRINT-based projects, because logs will be only reviewed by developers and system administrators using provided UI tools. We decided to use ElasticSearch in the AI-SPRINT Monitoring Subsystem as a data storage and search engine.

Pros:

- Written in Java, can be run on the SCONE platform;
- Focused on text log storage and analysis;
- Provides custom query language;
- Provides full-text search capabilities;
- Open source;
- Various deployment options on K8s (as StatefulSet, as K8s operator).

Cons:

- Suggested deployment configurations will need at least 3 K8s nodes (resource consuming);
- Influence of dual license is not clear for developers (and may be harmful for licensing software providing ElasticSearch as a SaaS search solution).

A summary of the evaluation process is presented in Table 3.8:

webpage	https://www.elastic.co/elasticsearch/
sources:	https://github.com/elastic/elasticsearch
license	Side Public License v1 and the Elastic License 2.0
module type	data storage
evaluation grade	very useful;

Table 3.8 ElasticSearch evaluation summary

3.2.4 Grafana

Grafana is a web application focused on data visualization and analysis. It can fetch time series and text data from various data sources. It contains dedicated plugins which allow fetching data from InfluxDB and ElasticSearch. Grafana can visualize data on configurable dashboards in the form of various widgets (graphs, gauges, tables). The possibility of integration of multiple data sources on a single dashboard makes it a very useful tool for system administrators and users.

We checked Grafana towards required data processing features. InfluxDB UI provides an advanced data exploration and visualization module but it cannot display text log data and does not provide easy aggregation data from other databases. Grafana fulfils both these requirements. It also introduces separation of concerns; users will not have to have accounts in the InfluxDB database in order to use dashboards. We noticed that it is more difficult to write data queries in Flux (InfluxDB query language) during dashboard definition. Fortunately this task will be performed rarely and we can propose the following workflow: user, who will be responsible for creating a new, custom dashboard, should have r/o account in InfluxDB and create Flux query in InfluxDB UI and then copy it (as text) and use in Grafana.

Grafana fulfils all requirements in the data presentation area, so we plan to use Grafana as a general-purpose visualization tool in the AI-SPRINT Monitoring Subsystem.

Pros:

- Configurable dashboards;
- Can load data from many data sources (including InfluxDB and ElasticSearch);
- Can present aggregated data from different data sources on one dashboard;
- Written in Go, can be run on the SCONE platform;
- Can be deployed on the K8s cluster; Helm charts available.

Cons:

- It is difficult to write and test Flux queries in InfluxDB data source definition.

A summary of the evaluation process is presented in Table 3.1Table 3.9:

webpage	https://grafana.com/
sources:	https://github.com/grafana/grafana
license	AGPL-3.0 License
module type	data visualization
evaluation grade	very useful;

Table 3.9 Grafana evaluation summary

4. Proposed architecture and initial implementation

After the analysis of requirements and mapping them to available open source software, we defined the architecture of the AI-SPRINT Monitoring Subsystem. It is assumed that this subsystem will work on and integrate with Kubernetes clusters.

The whole subsystem will be divided into three parts:

1. **Backend**: responsible for data storage and data processing, it will be based on InfluxDB 2.0.7 (metrics) and ElasticSearch 7.15.0 (logs);
2. **Data gathering components**: which will fetch and receive metrics, that will be developed by relying on Telegraf 1.14;
3. **Data visualization services**: which will provide detailed, visual presentation of gathered metrics and aggregated data (based on Grafana 8.2.2).

All these components will be deployed on the Kubernetes cluster hosting both the AI-SPRINT runtime and the running applications. The results of the analysis of the deployment options will be reported in the following sections providing the description of each part of the system. First in Section 4.4.1, we describe the proposed architecture of the AI-SPRINT Monitoring Subsystem. Then Section 4.4.2 provides details of each submodule: backend, data gathering and data visualisation. Section 4.4.3 hierarchical deployment to support metrics data storage. Finally Section 4.4.4 discusses the scalability options which can be exploited by the Monitoring Subsystem.

4.1.1 Proposed architecture

A typical Monitoring Subsystem deployment at runtime will look as depicted in Figure 4.1:

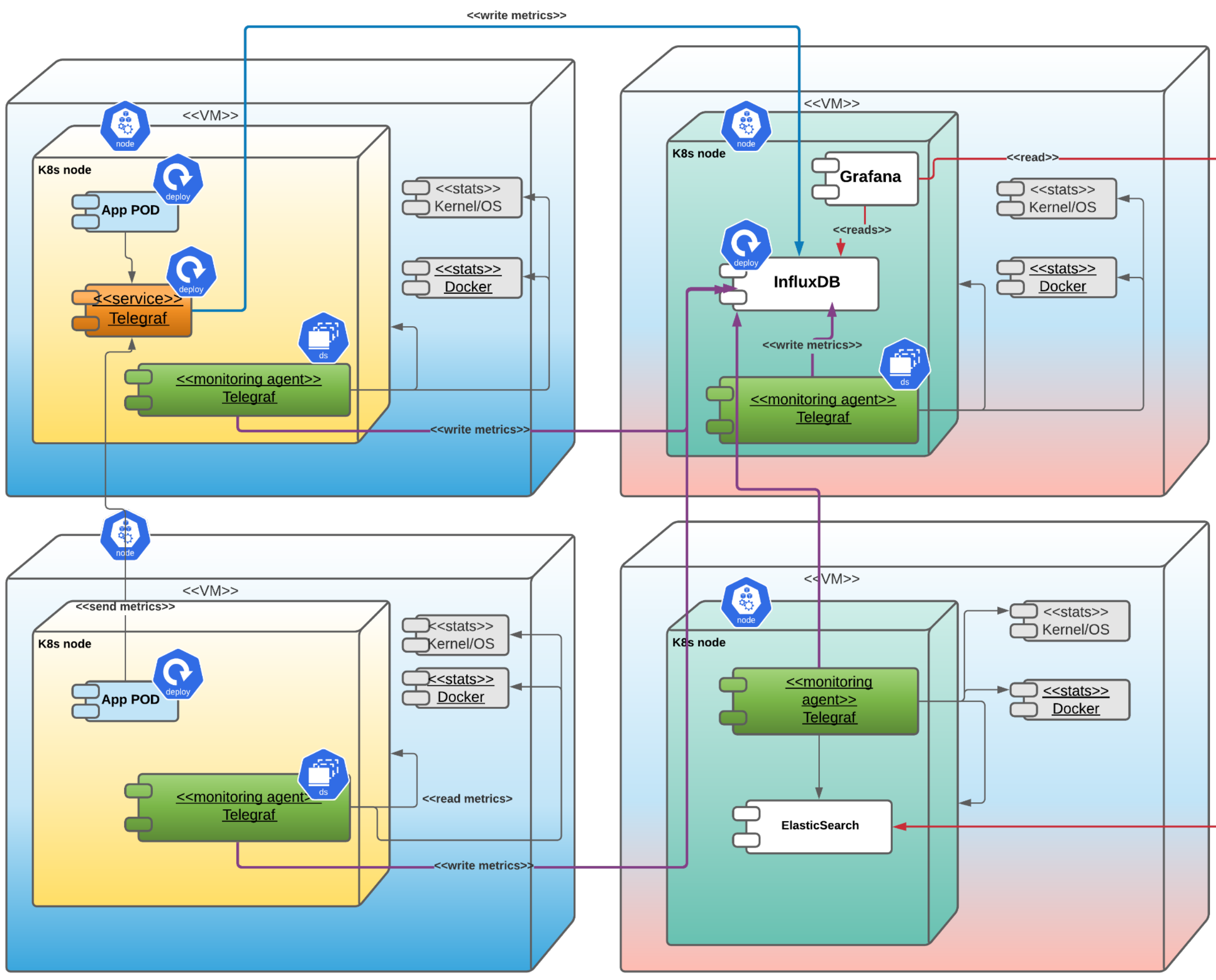

Figure 4.1 Monitoring Subsystem architecture

A monitoring subsystem instance will consist of a set of four main components:

- Time series database InfluxDB (backend) will be responsible for data storage and data query execution. It will also provide UI for administrators and REST API to simplify administration tasks.
- ElasticSearch (backend) will be responsible for providing data storage and search engine for text logs.
- Grafana (data visualization) will provide an UI for system users and administrators with dashboards showing various system statistics. Out of the box Grafana deployed in the Monitoring Subsystem will contain a predefined dashboard displaying a basic set of collected system statistics and logs. As this tool allows users to configure customized dashboards, adjusted to their needs, users will be able to modify or create new dashboards to present specific data. It is worth mentioning that Grafana has built-in support for various data sources (databases, systems) including InfluxDB and ElasticSearch. This will allow users to extend its capabilities according to their future needs.
- Telegraf (data gathering) will be responsible for gathering data and sending it to the InfluxDB database. The data gathering model will be a “pull” for system statistics, which will be fetched periodically (e.g., once every 5 seconds) and a “push” for monitoring short-lived tasks. In case of long-living applications we suggest using “push” or/and “pull” according to the developer’s needs.

4.1.2 Backend

InfluxDB

For the implementation of the Monitoring Subsystem backend, we have chosen InfluxDB as the database which will store and process the gathered system and application level metrics. As exposed in Section 3, InfluxDB is a universal time-series database, focused on processing large volumes of measured data. It provides an elastic data model, which allows storing metrics with additional metadata. The default database deployment also contains a set of useful tools, which ease administration and data management. This database fulfills project requirements for project data storage and data processing capabilities.

InfluxDB separates the stored data to one or more “organizations”. Each of them contains “buckets” where metric data is stored.

InfluxDB can authenticate clients (1) by username and password or (2) by token. We plan to use the former method to authenticate administrators (humans) and the latter for other software components which use the database's REST API to access data. For each token one can define access control rules which describe its read and write access rights. It is possible to create only “write” access permissions for a token. We plan to use such permissions for tokens generated for data gathering components.

This database also provides an alert management module. It can send messages to external applications when defined conditions are met. These conditions are defined in alerting rules, which are applied on incoming data streams. The Monitoring Subsystem will send HTTP POST requests to a given, external endpoint with alert messages encoded as a JSON structure.

Hierarchy support

It will be possible to build a hierarchy of InfluxDB databases. The “lower level” will be able to send metrics data to the “upper” one according to the given diagram reported in Figure 4.2. It states that lower levels will be at edge nodes while higher in the cloud and they need to be reconciled in case of network disconnections.

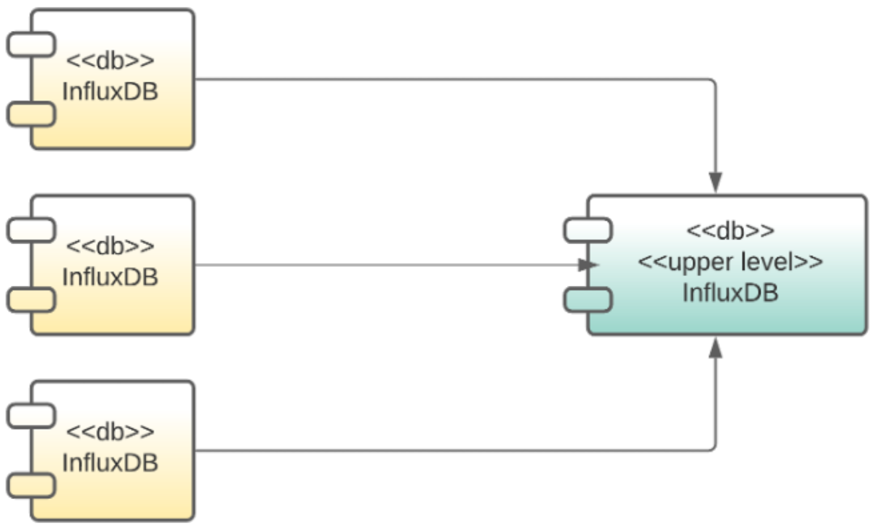

Figure 4.2 InfluxDB hierarchy support

It is also possible to fetch data from the lower level by the upper one, but the previous way is preferred.

In order to create such a hierarchy one will need:

1. Create “upper level” and “lower level” InfluxDB instances (as an element of the Monitoring Subsystem or as an independent installation).

2. The database which will be "upper level" should provide "write only" tokens for each of "lower level" instances. This will allow pushing data from the "lower level" to the "upper" one.
3. Administrator should define "connection" between "lower" and "upper" InfluxDB instances. We will create a special tool to manage such connections. This tool will provide periodic synchronization and will properly handle network issues (e.g. when one of the "lower level" InfluxDBs will be off-line for a long time).
4. At any moment the administrator will be able to delete such "connection".

ElasticSearch

The ElasticSearch database will be responsible for storing, indexing and providing log search capabilities. As it may require a significant amount of resources (CPU, RAM) it should be deployed on the separane K8s node, in order not to interfere with other running applications. As it will provide non-critical functionalities, we suggest creating a simple one-instance deployment. In case of any problems, the instance will be restarted or recreated on a different K8s node.

4.1.3 Metrics data gathering components

We will use Telegraf as a main component responsible for metric data gathering. Telegraf, as explained in Section 3, is a universal tool which can scrape metrics (pull) and receive metrics from monitored applications. It contains a broad set of input and output plugins, which makes it a very elastic and configurable tool. Writing data to the InfluxDB database is supported out of the box and also InfluxDB provides automatic generation of configuration files for Telegraf instances (which may be helpful for administrators). As its purpose is only to gather data and send it to the database, it has rather small memory and CPU requirements. Telegraf is an open source application written in the Go language. It can be run on the SCONE platform.

Each Telegraf will be authenticated in InfluxDB by a token, which will grant "write" permissions to a specific bucket in the database. This will increase security.

Monitoring agents

Monitoring Subsystem installation will contain at least two sets of Telegraf instances. The first (and basic one, depicted in green in Figure 4.1) will be "monitoring agents". They will be deployed on the Kubernetes cluster as a DaemonSet (on each K8s node). Each of them will be responsible for gathering metrics and statistics from each K8s node and VM hosting that node. Each Telegraf will send gathered data directly to InfluxDB. Each of these agents will only read (scrape) data and will not provide any network endpoint allowing it to receive data from external clients. This will increase security, because these Telegraf instances may have access to fragile infrastructure data.

By default "monitoring agents" will be collecting data using the input plugins described in Table 4.1.

Plugin name	**Short description**	**Documentation url**
cpu	CPU statistics fetched from the node operating system	https://github.com/influxdata/telegraf/tree/master/plugins/inputs/cpu

disk	Disk usage statistics fetched from the node operating system.	https://github.com/influxdata/telegraf/tree/master/plugins/inputs/disk
diskio	Disk I/O usage statistics fetched from the node operating system.	https://github.com/influxdata/telegraf/tree/master/plugins/inputs/diskio
kernel	Node's kernel statistics.	https://github.com/influxdata/telegraf/tree/master/plugins/inputs/kernel
mem	Node memory usage statistics	https://github.com/influxdata/telegraf/tree/master/plugins/inputs/mem
processes	Aggregated metrics describing processes running on the node	https://github.com/influxdata/telegraf/tree/master/plugins/inputs/processes
swap	Statistics of the "swap" fetched from the node's operating system.	https://github.com/influxdata/telegraf/tree/master/plugins/inputs/swap
system	Various metrics describing node's operating system statistics	https://github.com/influxdata/telegraf/tree/master/plugins/inputs/system
docker	Statistics reported by docker.	https://github.com/influxdata/telegraf/tree/master/plugins/inputs/docker
kubernetes	Kubernetes statistics (for pods and nodes);	https://github.com/influxdata/telegraf/tree/master/plugins/inputs/kubernetes
network	Network statistics fetched from the node operating system.	https://github.com/influxdata/telegraf/blob/master/plugins/inputs/net/NET_README.md

Table 4.1 Monitoring Subsystem architecture

These plugins will allow monitoring important parts of the whole system, like Kubernetes pods and nodes together with statistics from the Linux operating system which hosts Kubernetes nodes.

Monitoring metrics will be defined in an external YAML file. It will be provided by Application Manager and parsed by an ad-hoc tool to establish the configuration and to deploy the “monitoring agents” (Telegraf instances on each of K8s nodes). The configuration file will have the following structure:

```
monitoring: # (1)
  enabled-metrics: # (2)
  - disk
  - kubernetes
  - processes
  timePeriod: 5s # (3) metrics will be fetched once 5s
  nodes: # (4)
  - name: <node-name> # (5)
    enabled-metrics: # (6)
    - kubernetes
    - cpu
    timePeriod: 10s # (7) metrics will be fetched once 10s for this node
```

The “monitoring” (1) section contains all parameters needed to properly configure metrics collected by the Monitoring Subsystem. There will be a set of input plugins (see Table 4.1) which will be enabled in all instances of the “monitoring agents” (2) along with the “timePeriod” defining data gathering frequency (3). If there are nodes which will need a special set of input plugins, then there will be the possibility to define a separate set of parameters for each of them in the “nodes” section (4). Nodes will be identified by their name (5). And for each of them there will be defined “enabled metrics” (6) and “timePeriod” (7). These parameters will override default ones defined in (2) and (3).

If no configuration file is provided, the Monitoring Subsystem will be deployed and started with a default configuration containing common input plugins.

Telegraf services

The second set of Telegraf instances (“orange” in the diagram of Figure 4.1) will be deployed as a service and will allow applications to send their custom metrics to the network interface (TCP or UDP port) provided by Telegraf. It will buffer data locally and then send it to the InfluxDB database. The configuration of Telegraf will be minimal: it will contain only the network interface definition and definition of connection to the InfluxDB.

The usage of two separate sets of Telegraf instances will consume a little more resources, but it has a few advantages:

- “monitoring agents” will have access to crucial data on the node and VM (docker socket, files mounted from VM), so they should be separated as much as possible from other applications. Also their configuration usually will differ from Telegraf service’s used by AI-SPRINT applications (different default tags, input plugins).
- The Telegraf service will be able to send the collected data not only to InfluxDB but also to other destinations. It can also send data to different buckets in InfluxDB. Administrators can adjust its configuration without worrying about possible impact on monitoring data on one or more nodes. This improves the stability of the whole solution.
- It will be possible to create separate Telegraf services and adjust their configuration to the requirements of a monitored application. For instance, Telegraf may need to scrape (pull) metrics provided by the application.
- It will be possible to adjust security and authorization mechanisms to the monitored application. It will be possible to create K8s network policies allowing access to Telegraf service only for a selected set of applications.

4.1.4 Log data gathering components

In case of log data gathering, two scenarios will be supported: text logs will be loaded either from provided text files or from running applications. The first scenario will be used in situations, when there are many short-lived tasks or when logs were collected on edge devices. It is assumed that one of the other AI-SPRINT mechanisms will manage log files and send them to the log “import area”. Then the log import submodule will import it into the AI-SPRINT Monitoring Subsystem. This data flow is depicted in Figure 4.3.

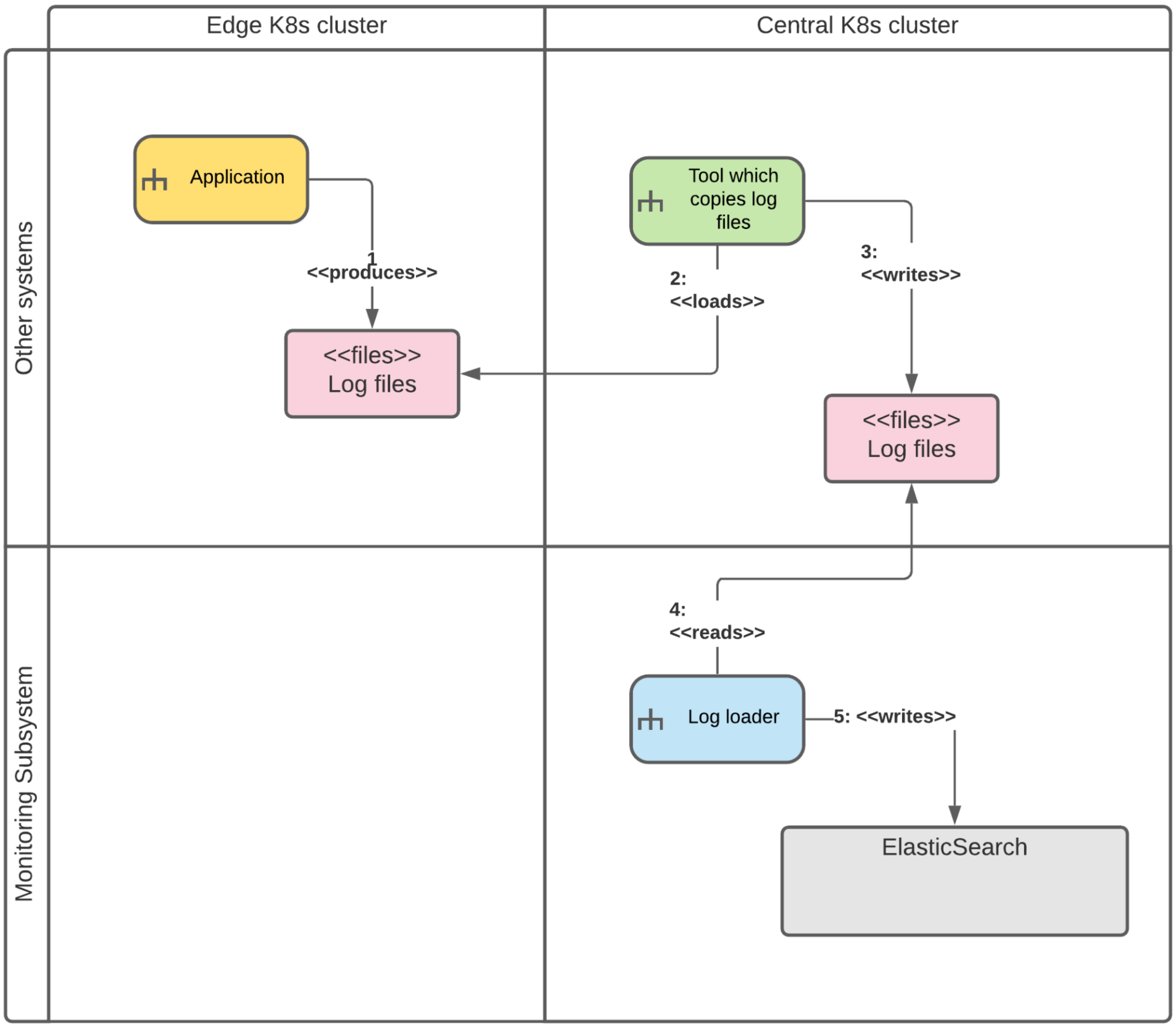

Figure 4.3 Importing logs into ElasticSearch

The second scenario will be used when logs are generated by long-lived applications. We plan to use one of ElasticSearch data importing modules (logging modules) such as FileBeat. They will be deployed automatically next to the running application instance and it will collect logs in real time and then send them to the ElasticSearch instance. This scenario is shown in Figure 4.4. Full logs gathering workflow will be implemented in M24 version of the Monitoring Subsystem.

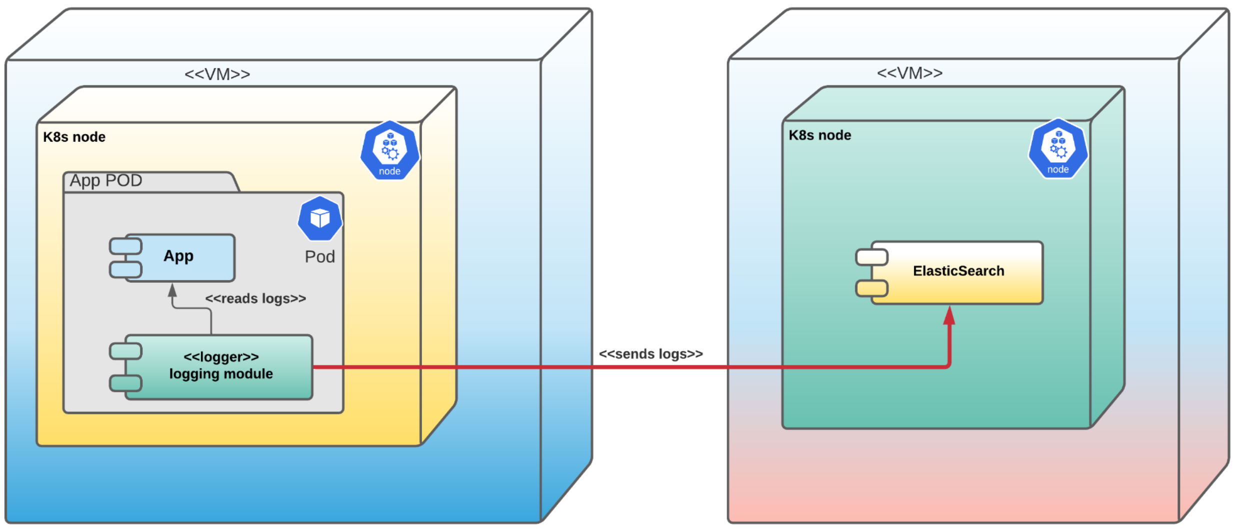

Figure 4.4 Sending logs fro running application to ElasticSearch

4.1.5 Data visualization

We plan to use Grafana as the main module which will provide visualization of data collected by the Monitoring Subsystem. As described in Section 3, Grafana is a popular tool used for data visualization purposes. InfluxDB contains a web UI interface, however it provides much less functionalities than Grafana. Also, Grafana allows to display data from various data sources (databases) which may be helpful in future, non-standard deployments.

We will provide a default dashboard which will display basic information about the monitored system state. This dashboard will fetch collected metrics from InfluxDB and display them as adjustable graphs. We assume that administrators (and other Grafana users) will adjust or create new dashboards fitting to their needs. Figure 4.5 shows the relationship between Grafana and InfluxDB (which is used as a data source in Grafana).

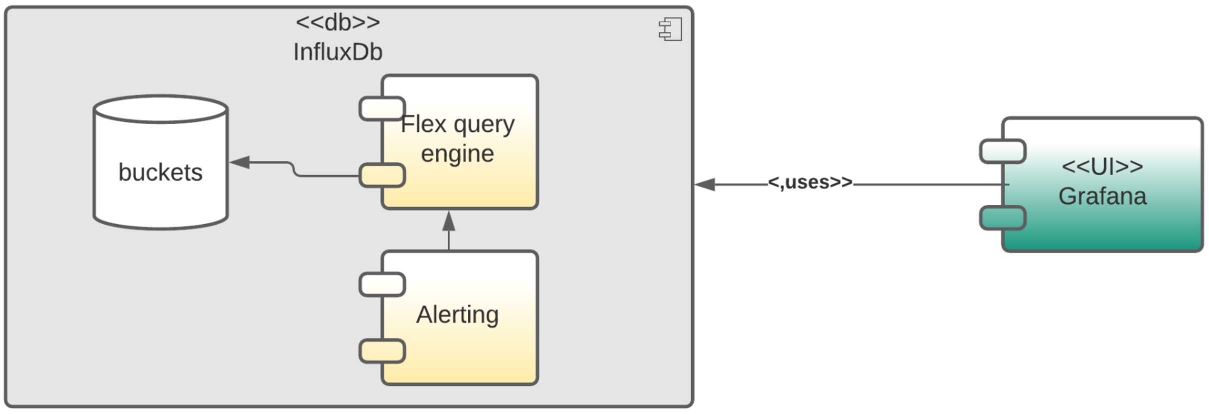

Figure 4.5 Grafana and its datasources

Grafana is an open source application and can be run securely on the SCONE platform.

4.1.6 Client libraries

It is assumed that applications will either send their custom metrics to the Telegraf service (push metrics) or provide them to be read (scraped) by Telegraf as a web page. Both approaches need to be supported by additional client side libraries in order to help software developers. There are many client libraries for Telegraf written for various programming languages. As AI-SPRINT will be mainly based on the Python language, we chose Pytelegraf (https://github.com/paksu/pytelegraf) library to be used by developers and in our tools.

If the application will need Telegraf to scrape metrics, we suggest that developers use the "Prometheus Client" library (https://github.com/prometheus/client_python). This library allows exposing application statistics over HTTP on a specified port. Then Telegraf, which has support for reading input data for Prometheus, will be able to read these statistics and send them to the InfluxDB instance.

According to requirements, applications must have the ability to define alerting rules in the Monitoring Subsystem. If the Monitoring Subsystem detects breaking these rules, it will have to send an alert message (data structure) to a dedicated module which will be responsible for processing these alerts.

Figure 4.6 describes how the client library will interact with components of the AI-SPRINT Monitoring Subsystem.

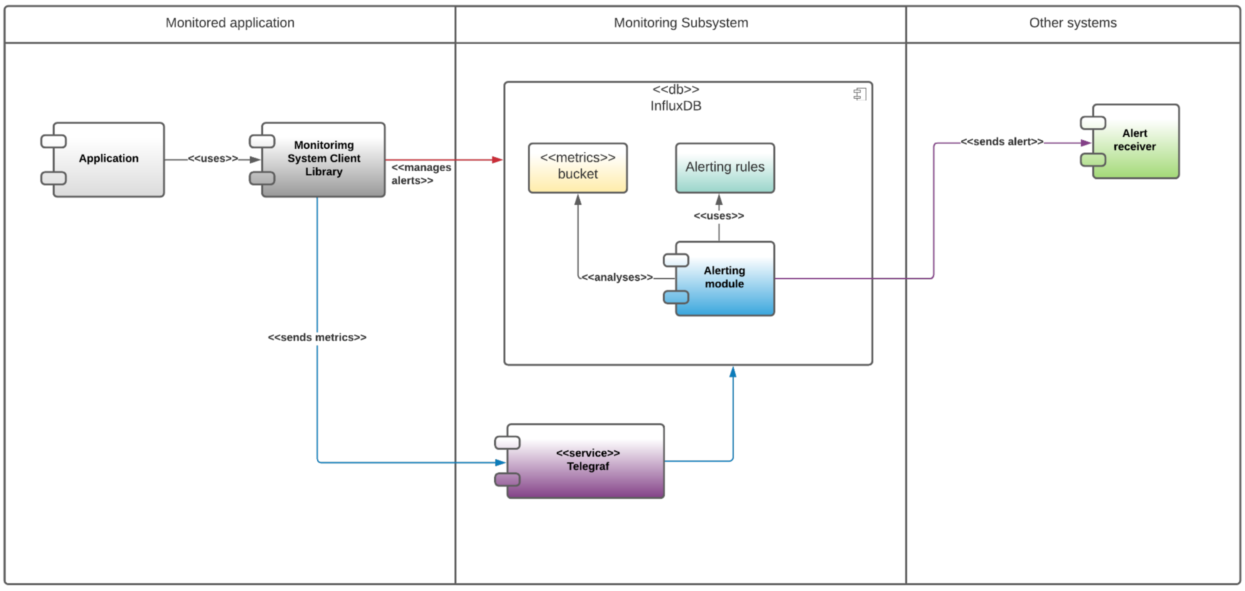

Figure 4.6 Monitoring System Client Library usage context

This module will provide a HTTP endpoint which will "consume" these alerts (it will accept POST requests containing alert messages in JSON format). There will be one web endpoint responsible for processing these alerts and implementation of such an endpoint (module) is out of the scope of the Monitoring Subsystem.

The Monitoring Client Library should provide functions which can be used to send metrics from annotations (decorators) handling Python functions. The required API is described in the AI-SPRINT Deliverable *D2.1 First release and evaluation of the AI-SPRINT design tools*. In general the Library will hide all sending metrics and alerts management and provide a simple Python API for developers. It will be integrated with Python wrappers (annotations) proposed in the D2.1 Deliverable. We will provide the function "to_monitoring_tool" which will send program execution time to the Monitoring Subsystem from the "@execTime" wrapper function body[1]. This functionality will be fully implemented at M24.

[1] See: D2.1 First release and evaluation of the AI-SPRINT design tools, Section 4.4.1 and Figure 4.5.

4.2 Monitoring Subsystem scalability

This section will describe how the elements of the Monitoring System can be scaled. The most important part of this Subsystem is the InfluxDB database. Only its enterprise, commercial version (https://www.influxdata.com/products/influxdb-enterprise/) supports clustering, HA and easy horizontal scaling. However, free of charge, the OSS version can provide limited High Availability and horizontal scalability. The HA can be provided by mirroring data writes using the "Influx Relay" application (https://github.com/influxdata/influxdb-relay). According to the documentation it "adds a basic high availability layer to InfluxDB" and "with the right architecture and disaster recovery processes, this achieves a highly available setup". The architecture of this solution is shown on Figure 4.7.

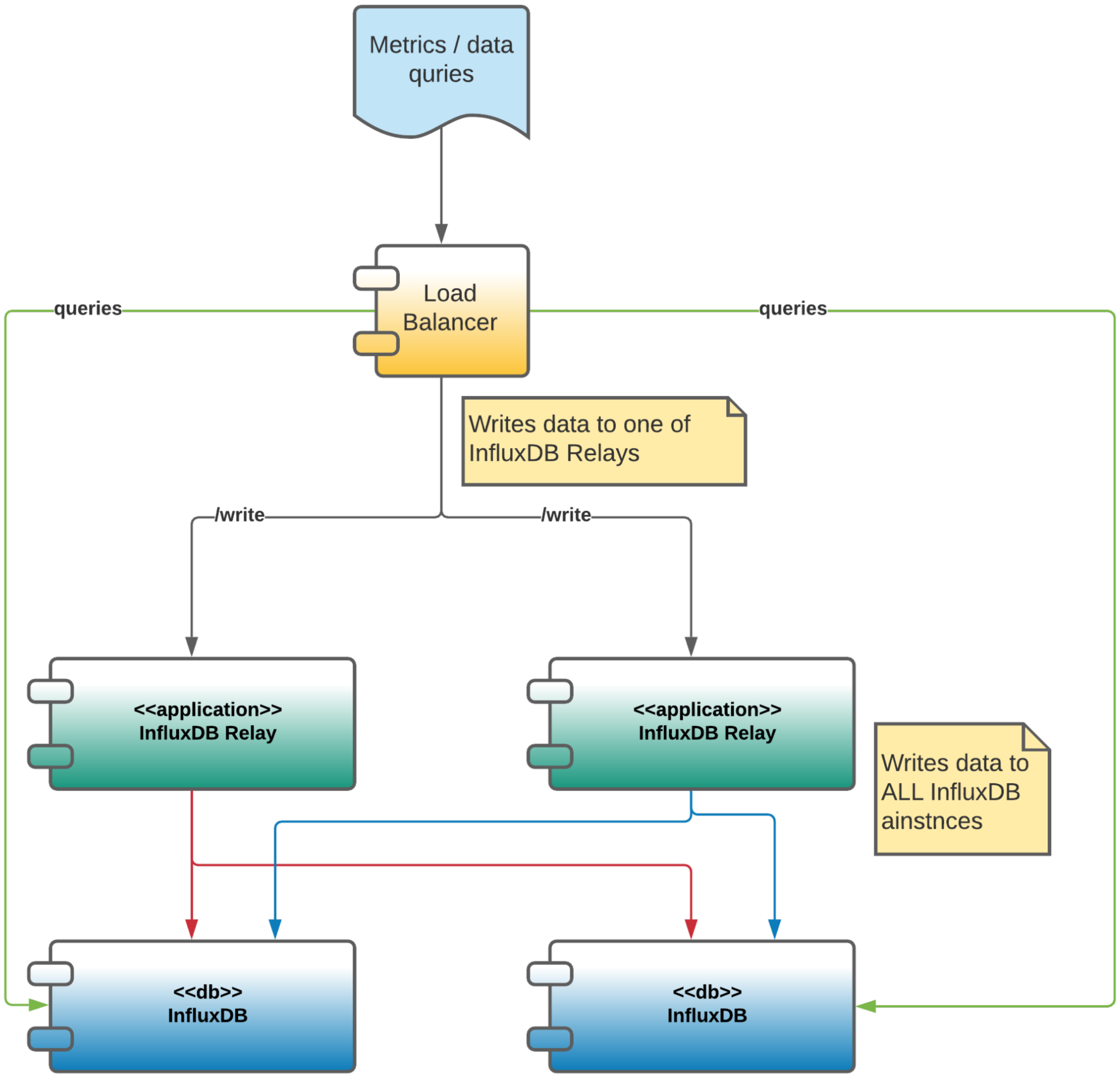
Metrics / data quries
Load Balancer
queries
queries
Writes data to one of InfluxDB Relays
/write
/write
<<application>> InfluxDB Relay
<<application>> InfluxDB Relay
Writes data to ALL InfluxDB ainstnces
<<db>> InfluxDB
<<db>> InfluxDB

Figure 4.7 InfluxDB data replication with InfluxDB Relay

Unfortunately, the setup of such configuration can be difficult to automate and the data recovery process is very complicated (and hard to be automated).

The preliminary analysis shown, that typical AI-SPRINT-based system installation will contain less than 10 application nodes. As there are no strong requirements for InfluxDB HA and scalability, we propose to set up a simpler data storage configuration for the OSS InfluxDB version. There will be one InfluxDB instance which will store data on network shared K8s Persistent Volume (PV). When this instance fails, a new one will be started and attached to the shared PV. Data loss probability can be minimized by buffering outgoing data in Telegraf instances. Also query performance and data processing can be improved by creating a group of read only InfluxDB instances which will download data from the master instance.

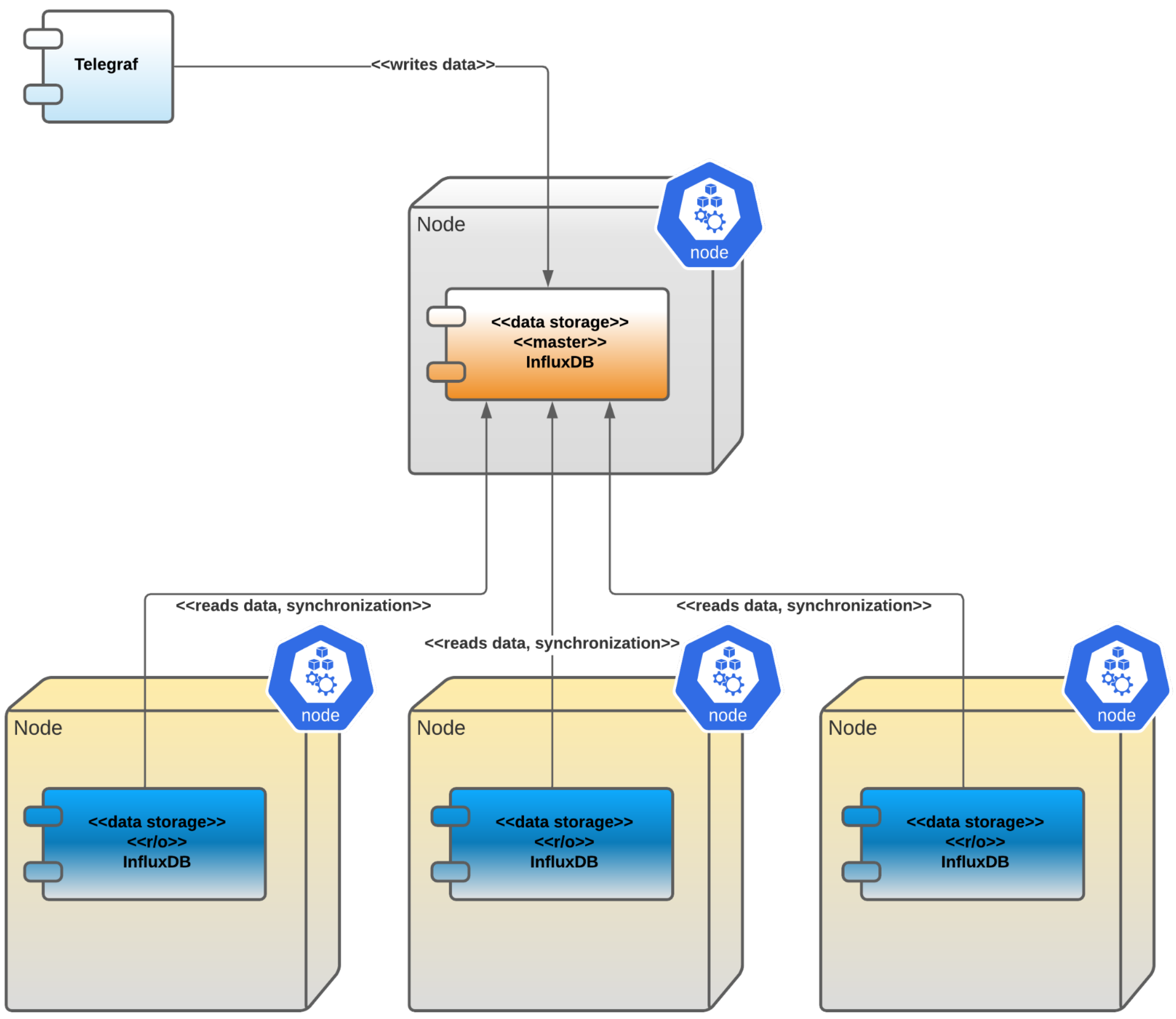

Figure 4.8 InfluxDB databases optimized for data analysis

This concept (depicted on Figure 4.8) is similar to building hierarchical structures of databases. These r/o databases will be used for performing heavy analytics and resource consuming queries.

Grafana - the application for data visualization - can be easily scaled horizontally. In such configuration it will need a scalable RDBMS such as MySQL or PostgreSQL. Also advanced caching modules (Redis or Memcached) should be provided. In case of a typical AI-SPRINT installation, a single instance of Grafana should be sufficient. However it can be horizontally scaled if necessary.

Telegraf monitoring agents are deployed on the K8s cluster as a DaemonSet. Every K8s node is running a single Telegraf instance. There is no requirement to provide scalability and high HA in this area. In case of any problems, the cluster will restart this monitoring agent.

Telegraf services will be able to be easily scaled using standard mechanisms provided by Kubernetes. We suggest that in typical cases, there should be provided two replicas running on separate nodes. The Kubernetes service mechanism will be responsible for distributing incoming network requests among these instances. In case of scraping metrics from running applications, we suggest using either the Telegraf Operator (which installs a sidecar next to each monitored container) or provide a single separate Telegraf POD which will scrape all services. This will prevent unneeded data duplication issues.

5. Monitoring subsystem deployment

In this section we describe how to deploy the AI-SPRINT Monitoring Subsystem. All components of the Monitoring Subsystem will be deployed on the Kubernetes cluster. We assume that there will be enough K8s nodes to provide basic cluster functionalities. We suggest that the core Monitoring System infrastructure (backend and data visualization) should be deployed on dedicated nodes. Telegraf instances being "monitoring agents" must be deployed on each node in order to provide the monitoring of the whole cluster. We also suggest that ElasticSearch and data importing modules should be deployed on a separate K8s node in order not to interfere with other running applications.

The deployment of the AI-SPRINT Monitoring Subsystem will be automatically performed by IM according to the deployment procedures described below. The UPV team will develop a recipe with all the installation procedures described in TOSCA language. They will be then automatically translated by IM into Ansible rules and executed on a given cluster infrastructure.

As general dependencies, the deployment procedure requires that kubectl and Helm tools are available and properly configured. Kubectl should allow full administrative access to the Kubernetes cluster.

All commands must be executed in a Linux terminal. It is assumed that the Monitoring Subsystem will be deployed in its own K8s namespace called "ai-sprint-mon". Telegraf services responsible for gathering data from applications should be deployed in the same namespaces as monitored applications.

Taking all these considerations into account, the deployment process is composed of five parts that are detailed below.

Part 1: prepare cluster

1. taint[2] all nodes for InfluxDB and Grafana with value: ai-sprint-mon=infrastructure:NoSchedule
2. label nodes for InfluxDB: ai-sprint-mon/nodetype: "db"
3. label nodes for Grafana: ai-sprint-mon/nodetype: "ui"

Part 2: deploy an InfluxDB instance

At first InfluxDB should be installed on the K8s cluster.

1. Configure access to Helm's "influxdata" repository:
   `helm repo add influxdata "https://helm.influxdata.com/"`
2. Execute:
   `kubectl create namespace "ai-sprint-mon"`
3. Adjust values describing InfluxDB initial credentials (admin username, admin user password, tokens) defined in "01-influxdb_secrets.yaml" file and then create these secrets in K8s cluster:
   `kubectl apply -f 01-influxdb_secrets.yaml -n "ai-sprint-mon"`
4. Adjust parameters describing expected InfluxDB deployment in "values" file for Helm tool: "01-influxdb_values.yml" and then execute:
   ```
   helm install -n "ai-sprint-mon" -f 01-influxdb_values.yml -o json influxdb
   bitnami/influxdb
   ```
5. InfluxDB should be available in a few minutes.

Part 3: configure InfluxDB instance

In the InfluxDB, the administrator must choose the Organization and bucket where metrics must be stored. Also must create access tokens for Telegraf instances.

[2] https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/ .

1. Administrator should log into a running InfluxDB instance and choose (or create a new one) Organization and create a bucket for data storage. It is also important to define a data retention policy for that bucket.
2. Administrator must create a new token with "write" rights to that bucket. This will be token for monitoring agents.

Part 4: deploy Telegraf "monitoring agents"

On Each K8s node must be deployed Telegraf playing the "monitoring agent's" role.

1. Adjust values in "telegraf-ds-config.yaml" file: provide valid organization name, bucket name and value of token for monitoring agents.
2. execute:
   kubectl apply -f "telegraf-ds-config.yaml" -n "ai-sprint-mon"
3. adjust parameters in "telegraf-ds.values.yaml" file and execute:
   helm install -n "ai-sprint-mon" telegraf influxdata/telegraf-ds --values "telegraf-ds.values.yaml"

Part 5: deploy Grafana

Finally one has to deploy and configure Grafana, which will provide data visualization. Use scripts and templates in "grafana" subdirectory:

1. Create grafana-secrets.yaml from grafana-secrets.yaml.template file. Adjust values of administration account (user name and its password);
2. Using grafana.values.yaml.template file create grafana.values.yaml file and adjust parameters for Grafana. Follow instructions which are placed in this file (in comments).
3. Execute script: deploy-grafana.sh which will deploy Grafana on the cluster.

After all instances of Telegraf are deployed, check if system statistics have started to appear in the InfluxDB database.

This procedure will provide a basic Monitoring Subsystem deployment which will monitor K8s cluster and nodes.

Setting monitoring of applications

In order to provide monitoring for applications, one needs to configure a new token with "write" permissions for the data bucket.

1. Administrator should log into a running InfluxDB instance and create a new token with "write" permissions for the used bucket.
2. Deploy Telegraf in the namespace used by the monitored application, adjust "telegraf.conf" and execute:
   kubectl create secret generic telegraf-secrets --from-file=conf=./telegraf.conf
   kubectl create -f telegraf_deployment.k8s.yaml
3. Application then can send data to the Monitoring Subsystem using apriopriate Telegraf client library (ex. Pytelegraf)

There is also another way to provide integration between the application and the Telegraf instance. One can use the Telegraf Operator tool to provide a Telegraf "sidecar" installed next to a monitored application container in the same POD. This will consume more resources (one application container will require one Telegraf container) but will also give better security control and the ability to define Telegraf configuration in the application's deployment descriptor. Unfortunately in the case of executing short-living containers (tasks) the startup time of the Telegraf instance may be longer than the startup time of the monitored application. This may cause loss of metrics sent before Telegraf readiness or task slow down because of handling network errors.

In order to provide Telegraf sidecar one needs:

1. Update Helm repository:
   helm repo add influxdata https://helm.influxdata.com/
2. Generate new "write" token in InfluxDB;
3. Adjust telegraf-operator_values.yml (“write” token value, org. name, bucket name, server address);
4. Deploy Telegraf Operator:
   helm upgrade -f telegraf-operator_values.yml --install telegraf-operator influxdata/telegraf-operator
5. Provide annotations in application deployment descriptor according to the documentation of Telegraf Operator project: https://github.com/influxdata/telegraf-operator.
6. Deploy application. In short time metrics sent by the application should appear in InfluxDB.

6. Monitoring subsystem evaluation

In order to evaluate the chosen technology and prove that the requirements analyzed in section 2 are able to be fulfilled, we created a set of small proof-of-concept projects, which are available in the AI-SPRINT project gitlab (https://gitlab.polimi.it/ai-sprint). This evaluation has allowed us to confirm and test the proposed solution presented in Section 4.

6.1 InfluxDB and Telegraf evaluation

At first we confirmed that it is possible to deploy and configure the InfluxDB database on the Kubernetes cluster using the Helm tool. Then we tested how to deploy Telegraf and send system metrics gathered by Telegraf to the InfluxDB instance. We checked how to apply good practices suggested by Kubernetes documentation, such as credentials stored in K8s secrets and contents of configuration files stored in K8s configuration map objects.

The source code is available at git@gitlab.polimi.it:ai-sprint/monitoring-poc1.git.

6.2 InfluxDB REST API evaluation

We tested functionalities provided by the InfluxDB REST API. We demonstrated its capabilities in a simple administration library PoC demo. This demo showed how to create new access tokens in InfluxDB. This functionality will be needed in future application deployment procedures. It will allow the creation of separate “write” tokens for specific applications or Telegraf services deployments. The source code is available at git@gitlab.polimi.it:ai-sprint/monitoring-poc1.git (“automat” subdirectory).

6.3 Alert evaluation

We created a PoC showing how to define alerts and receive alert notifications using features provided by InfluxDB. This functionality was demonstrated by using two simple Python applications. One was resonsible for sending custom metrics data to Telegraf, which then forwarded them to the InfluxDB database. InfluxDB had defined alerting configuration and was sending alert messages to the second Python test application using a HTTP POST request.

The source code is available at git@gitlab.polimi.it:ai-sprint/monitoring-python-alert-receiver-poc-1.git

6.4 Sending data between InfluxDB instances evaluation

The “Data processing” requirement illustrated in Section 2.2.3 “” describes the need to build hierarchies of Monitoring Systems, where one of them is able to fetch data from other instances and perform further analysis. We tried to provide such functionality using mechanisms available in InfluxDB. We successfully demonstrated that it is possible to fetch the data available in a bucket located in the remote InfluxDB instance using built-in functions available in the Flux language and asynchronous tasks. We created a special periodic task which was able to download new data from a remote influxDB instance. This task was also able to properly handle network failures. When data was not able to be downloaded, it was taken into account and the next task execution used proper timestamp filters to download data not fetched before.

The task code is presented below:

```
import "influxdata/influxdb/tasks"
import "array"

option task = {name: "Import aggregated", every: 1m}

timeIdx = 2021-01-01T00:00:00Z
lastTaskRuns = from(bucket: "taskruns")
    |> range(start: timeIdx)
    |> filter(fn: (r) =>
        (r["_task"] == task.name))
    |> last()
    |> findColumn(fn: (key) =>
        (key._field == "lastSuccess"), column: "_value")

option tasks.lastSuccessTime = if exists lastTaskRuns and length(arr: lastTaskRuns) > 0
then time(v: lastTaskRuns[0]) else now()

array.from(rows: [{
    _time: timeIdx,
    _field: "lastSuccess2",
    _task: "debug",
    _measurement: "m",
    _value: tasks.lastSuccessTime,
}])
    |> to(org: "test.org", bucket: "taskruns")

option lastSuccess = tasks.lastSuccess(orTime: -1m)

data = from(
    bucket: "aggregated",
    host: "http://influxdb.default:8086",
    org: "test.org",
    token: "INFLUXDB TOKEN",
)
    |> range(start: lastSuccess)

data
    |> to(org: "test.org", bucket: "remote")

rec = array.from(rows: [{
    _time: timeIdx,
    _field: "lastSuccess",
    _task: task.name,
    _measurement: "m",
    _value: now(),
}])

rec
    |> to(org: "test.org", bucket: "taskruns")
```

The source code of the PoC solution is located at git@gitlab.polimi.it:ai-sprint/monitoring-poc1.git in the "test_tasks/poc_working_solution" directory.

This PoC showed that it was possible to create a hierarchy of InfluxDB instances, however this solution is complicated and will be hard to maintain. This is why we plan to develop a special, dedicated tool responsible for metrics synchronization.

6.5 Telegraf Operator evaluation

We tested the ability to provide monitoring capabilities to existing applications with minimal effort from the programmer's side. The easiest approach relies on the Telegraf Operator. It provides easy integration among applications deployed on K8s and Telegraf instances. Software developers should only provide a few annotations in K8s deployment configuration file and it will automatically create Telegraf "sidecars" next to running application containers. The sample YAML file presented below contains such configuration (look at "annotations" section):

```
apiVersion: v1
kind: Pod
metadata:
 labels:
   project: AiSprint
   test: "true"
 name: python-metrics-job-poc2
 annotations:
   telegraf.influxdata.com/inputs: |+
     [[inputs.socket_listener]]
     service_address = "udp://localhost:8092"
     data_format = "influx"
   # this defines part of Telegraf configuration responsible for sending data to
InfluxDB (or other locations):
   # you can use any data entry in secrets def. "telegraf-operator-classes"
   telegraf.influxdata.com/class: infra3
   telegraf.influxdata.com/limits-cpu: '750m'
   # invalid memory limit, which will be ignored
   telegraf.influxdata.com/limits-memory: '800x'
spec:
 containers:
 - image: localhost:5000/python-metrics-job-poc2:latest
   imagePullPolicy: Always
   name: python-metrics-job-poc2
   resources: {}
 dnsPolicy: ClusterFirst
 restartPolicy: Never
```

In the given example, "telegraf.*" annotations describe parameters of the Telegraf sidecar container which must be created next to "python-metrics-job-poc2" container.

These "sidecars" can either scrape metrics or receive metrics sent by application to a fixed port on the "localhost" address. Application will only eiter to provide metrics on a web endpoint or use Telegraf client library to send metrics to fixed port on "localhost". With this approach, the integration will be easy, however in case of short-living Python tasks, preliminary analyses demonstrated that the initialization of "sidecar" took more time than the task and some metrics sent to the UDP port (the fastest solution) were lost and did not appear in the InfluxDB database. The only solution was to use Telegraf deployed as a "classic" K8s service.

The source code that supported this analysis is available at git@gitlab.polimi.it:ai-sprint/monitoring-poc1.git in test_python/python_metrics_poc directory.

6.6 Proposed architecture evaluation

We created documentation and a set of configuration files for Helm and kubectl which allow semi-automatic deployment of the Monitoring Subsystem. For test purposes we created a Kubernetes cluster on top of an OpenStack infrastructure. This cluster was created using the Infrastructure Manager (IM) service. Almost all elements were created without any problems, however we had to apply a small fix in order to have the Kubernetes metrics working properly.

The testing cluster infrastructure includes 9 nodes:

- 1 master node (2 vCPUs, 4GB RAM);
- 1 monitoring node for InfluxDB database (2 vCPUs, 8GB RAM);
- 1 monitoring node for Grafana visualization tool (2 vCPUs, 4GB RAM);
- 6 nodes for test applications (2 vCPUs, 4GB RAM);

To deploy the Monitoring Subsystem we used the developed configuration files and scripts described in Section 5. All required monitoring modules (database, visualization, monitoring agents) were installed in a separate cluster namespace ‘ai-sprint-mon”. We verified the correctness of the deployment procedure and scripts. We also checked which metrics gathered by “monitoring agents” worked as expected and present true statistics describing the state of cluster elements.

After all components were installed we were able to visualize gathered metrics in Grafana (data visualization module), see Figure 6.1

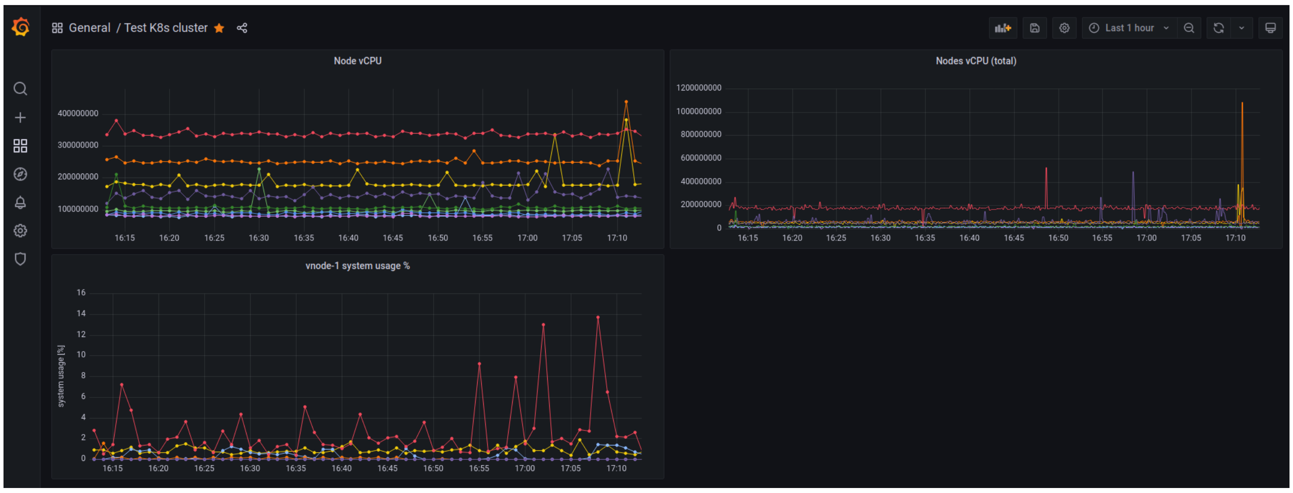

Figure 6.1 Data visualization in Grafana. Screenshot

The screenshot in Figure 6.1 shows a custom, default dashboard configured in Grafana, which presents system statistics collected from the test instance of the Monitoring Subsystem.

System administrators were also able to use UI provided by InfluxDB, to manage credentials (access tokens) and check values of gathered metrics, see Figure 6.2.

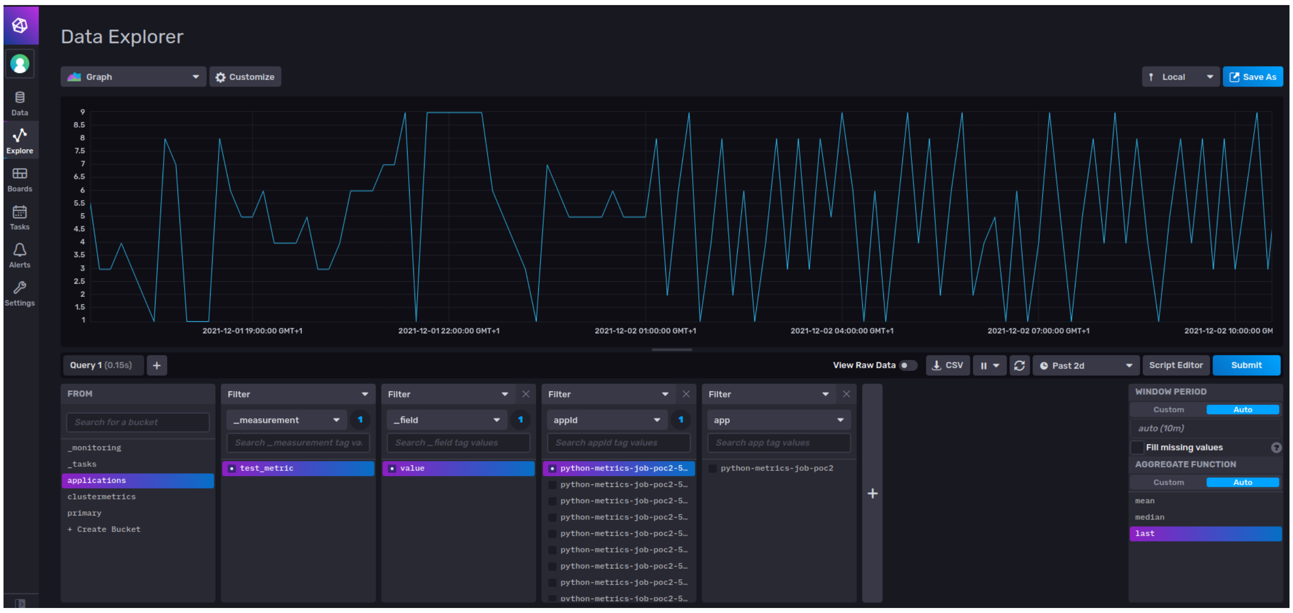

Figure 6.2 Gathered data can be viewed and analysed in web UI provided by InfluxDB. Screenshot

During tests we checked the quality of collected metrics. Most Telegraf's input plugins provided valuable metrics, but "net" plugin presented network statistics collected in the context of Telegraf container (and not at the node level, as expected). The source code of this analysis is available at git@gitlab.polimi.it:ai-sprint/monitoring-subsystem.git

6.7 Client library evaluation

During the evaluation phase we wrote a small PoC Python application which was using the "TelegrafClient" Python library to send metrics to InfluxDB via Telegraf instance. We deployed this sample application on K8s cluster and this library was working perfectly well.

```
import os
import time

from telegraf.client import TelegrafClient

telegrafHost = os.getenv("TELEGRAF_SVC")
if telegrafHost is None:
   telegrafHost = "localhost"

appId = os.getenv("APP_ID")
if appId is None:
   appId = "-"

nodeName = os.getenv("NODE_NAME")
if nodeName is None:
   nodeName = "-"

print("Starting... Data will be sent to telegraf on " + telegrafHost + "\n")

client = TelegrafClient(host=telegrafHost, port=8092)

# Records a single value with no tags
while True:
   for i in range(1, 10):
       client.metric('test_metric', i, tags={'app': 'python-metrics-job-poc2', 'nodeName':
nodeName, 'appId': appId})
       time.sleep(0.1)
   print("Metrics sent.\n")
```

In a final Monitoring Client Library all communication details will be hidden and provided out-of-the box by the library. Developers will only use a single line of code to write metrics to the Monitoring.

The source code of this PoC is available in GIT repository located at: git@gitlab.polimi.it:ai-sprint/python-metrics-job-poc2.git.

We also prepared the first, simple PoC version of the client library module which showed how to simplify and automate creation of alerts in InfluxDB from Python code. Thanks to such an approach, interaction with InfluxDB is very simple.

```
import os

from influxdb_client import InfluxDBClient

from alerts_lib import alerts

influxdb_addr = os.getenv("POC_METRICS_INFLUXDB_ADDR")
influxdb_token = os.getenv("POC_METRICS_INFLUXDB_TOKEN")
influxdb_org = os.getenv("POC_METRICS_INFLUXDB_ORG")
influxdb_remote_endpoint = "remote1"
influxdb_remote_addr = "http://notification-poc:8085/alert"

client = InfluxDBClient(url=influxdb_addr, token=influxdb_token, org=influxdb_org)

alerts_configurer = alerts.MonitoringAlertsConfigurer(client=client,
                                                       notif_endp_url=influxdb_remote_addr,

notif_endp_name=influxdb_remote_endpoint)

alerts_configurer.configure_alert(20.9)
```

The source code of this PoC is available in the GIT repository located at: git@gitlab.polimi.it:ai-sprint/python-metrics-job-poc1.git.

6.8 Performance tests

In order to check the performance of the Monitoring Subsystem deployed on the real Kubernetes cluster, we created a special test K8s environment. We used Cloud&Heat OpenStack cluster and IM to configure and run the test K8s cluster. This installation consisted of 9 nodes deployed on OpenStack infrastructure:

- 1 master node (2 vCPUs, 4GB RAM);
- 1 monitoring node for InfluxDB database (2 vCPUs, 8GB RAM);
- 1 monitoring node for Grafana visualization tool (2 vCPUs, 4GB RAM);
- 6 nodes for test applications (2 vCPUs, 4GB RAM);

We also created a simple test Python application which was sending a simple, custom numeric metric every 100ms to the Monitoring Subsystem using Telegraf service. We investigated the influence of test application instances count on InfluxDB performance. We were also interested in checking InfluxDB abilities to receive measured data from multiple monitoring agents and telegraf services.

During the tests we measured system load and resource consumption when the cluster was idle (no applications were monitored) and under a heavy load.

At first we investigated the system usage of the running InfluxDB container in the function of monitored applications count. This value was measured and provided by the Docker engine. Collected values are presented in Figure 6.3.

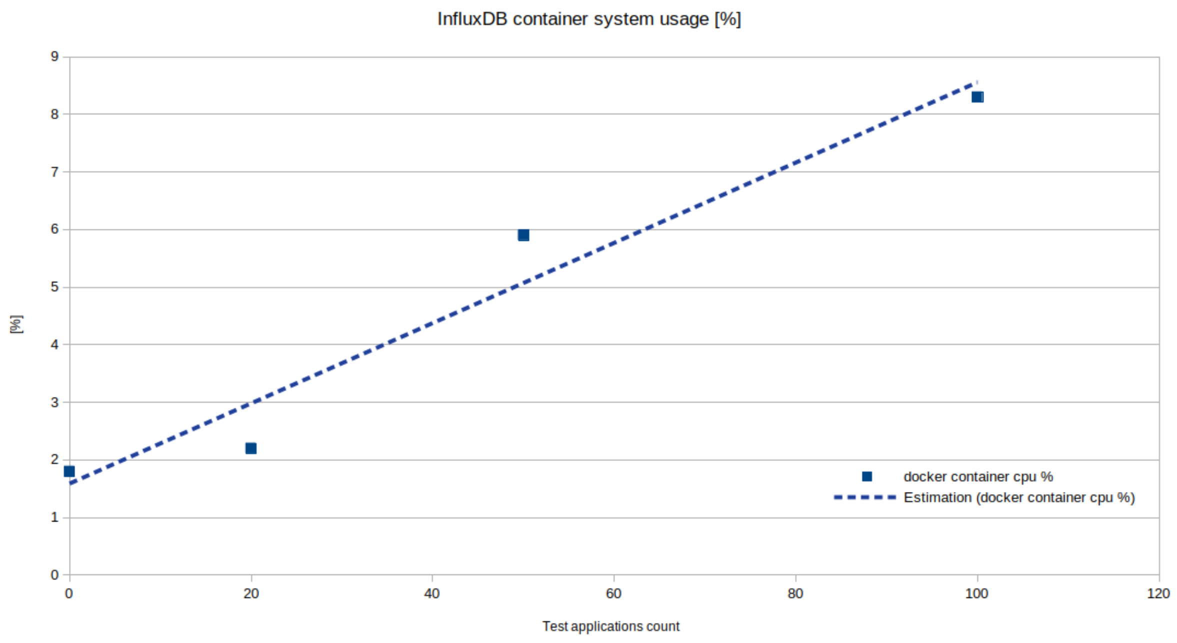

Figure 6.3 measured InfluxDB's container system usage [%]

We observed that InfluxDB consumed at about 2% system usage when it only received data from 9 monitoring agents. We tested the impact of increasing numbers of client applications on InfluxDB for 0 (no client applications), 20, 50 and 100 client applications simultaneously sending metrics data to the Monitoring Subsystem (and then stored in InfluxDB) via a single Telegraf service. We observed that there is almost linear correlation between the number of test Python applications and InfluxDB container system usage.

Then we measured "system load" reported by the operating system which hosts the Kubernetes node. For 0 (no client applications), 20, 50 and 100 client applications simultaneously sending metrics, we gathered average values of "1 minute load" and "15 minutes load" statistics.

Collected values are presented in Figure 6.4.

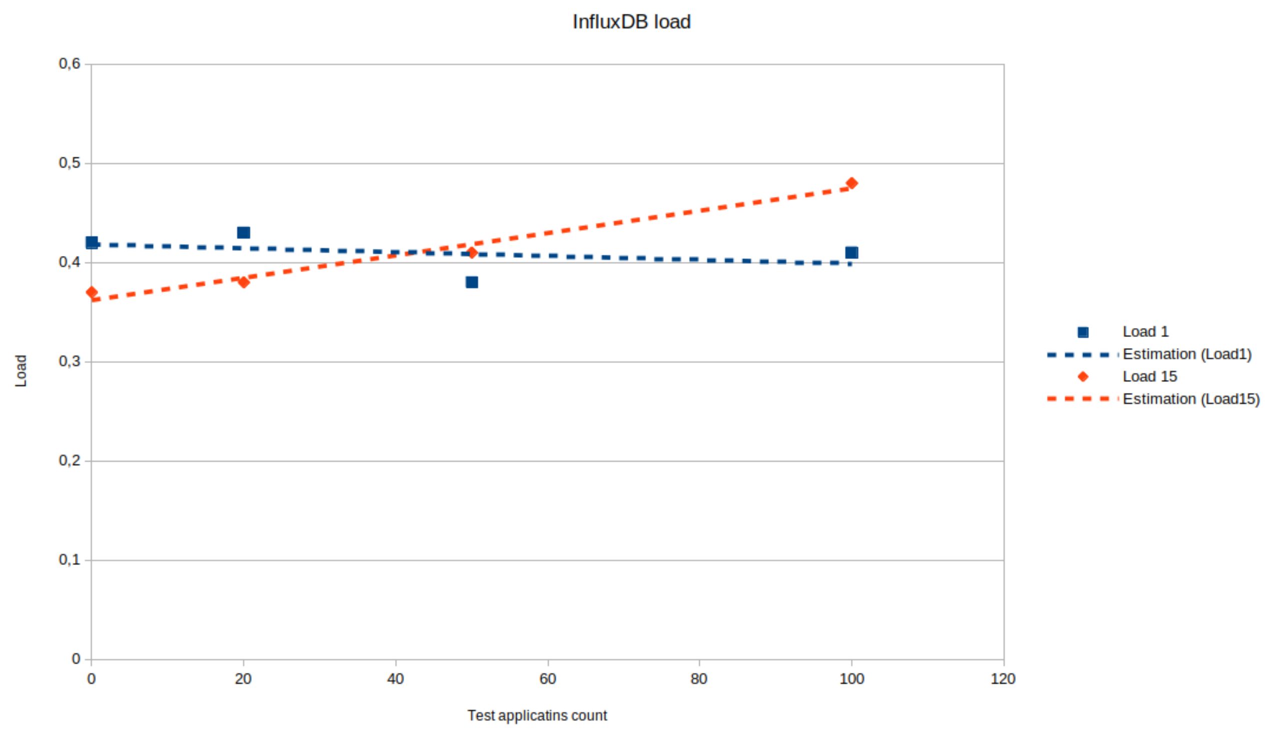

Figure 6.4 measured InfluxDB's node load (load 1m and load 15m)

We observed there was almost no influence of test applications on system load on the node where the InfluxDB instance is deployed.

We also measured vCPU resources consumption, measured in "nanocores"[3], for the whole K8s node and for InfluxDB container. These values were reported by the kubelet running on the node where InfluxDB was installed. We gathered these values for 0 (no client applications), 20, 50 and 100 client applications simultaneously, sending metrics. We observed an almost linear correlation between test applications count and vCPU consumed by InfluxDB and the whole node. Results are presented on the diagram in Figure 6.5:

[3] https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu

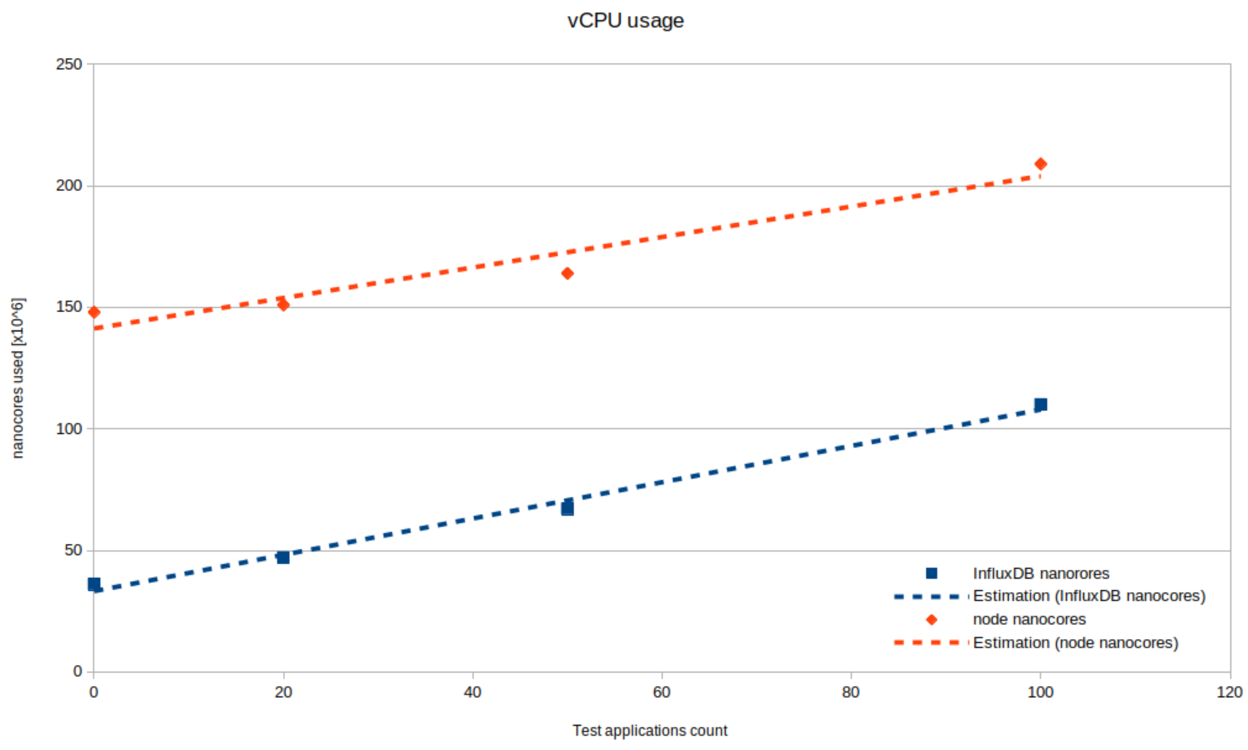

Figure 6.5 vCPU usage of InfluxDB container and whole node

Finally, we have also measured the influence of Kubernetes infrastructure monitoring on the InfluxDB performance. In all aboved tests, the gathering metrics from Kubernetes nodes, belonging to the cluster, was turned on. There were 9 nodes constantly measured by 9 Telegraf instances ("monitoring agents") running on each of these nodes. In order to measure the amount of resources which were consumed by idle ("isolated") InfluxDB, we uninstalled all these "monitoring agents" from the cluster. Then we measured the vCPU consumption and system usage. Results are shown on Figure 6.6 and Figure 6.7. The "isolated InfluxDB" graph represents a situation when InfluxDB was not used by any client. The "0" graph shows a case when InfluxDB received only data from 9 "monitoring agents" and there was no client application sending metrics data to the InfluxDB database. The impact of "monitoring agents" on InfluxDB performance was negligible.

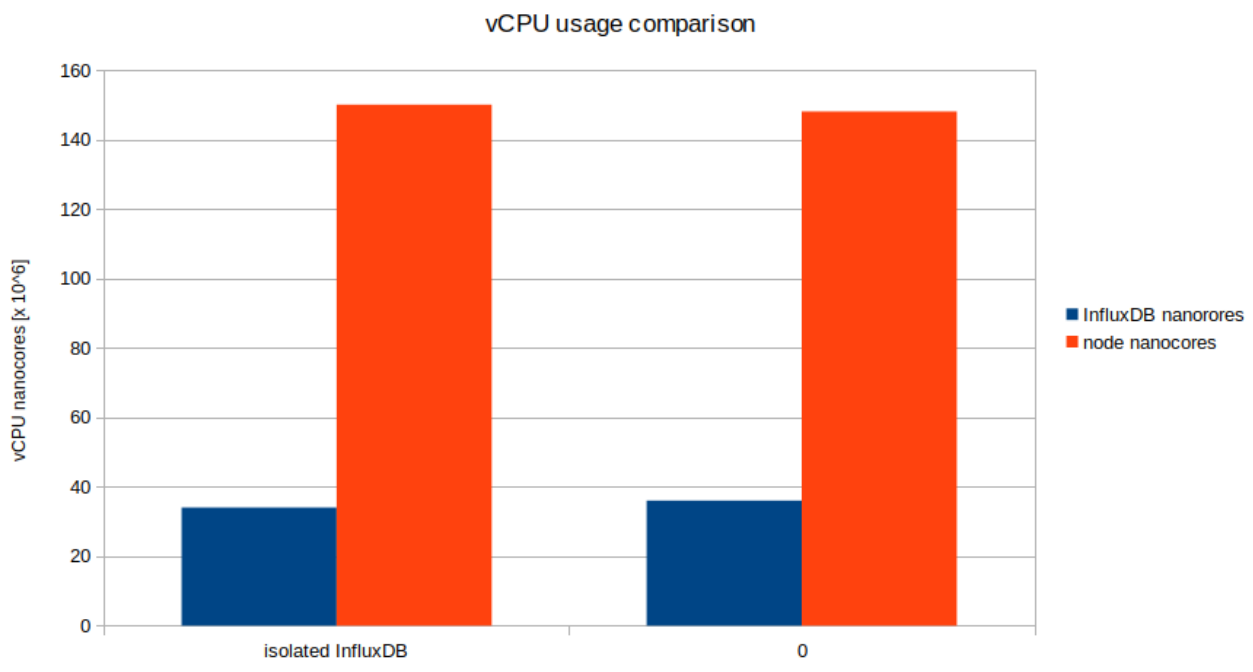

Figure 6.6 vCPU usage comparison: isolated (idle) InfluxDB and InfluxDB gathering data only from the 9 K8s nodes

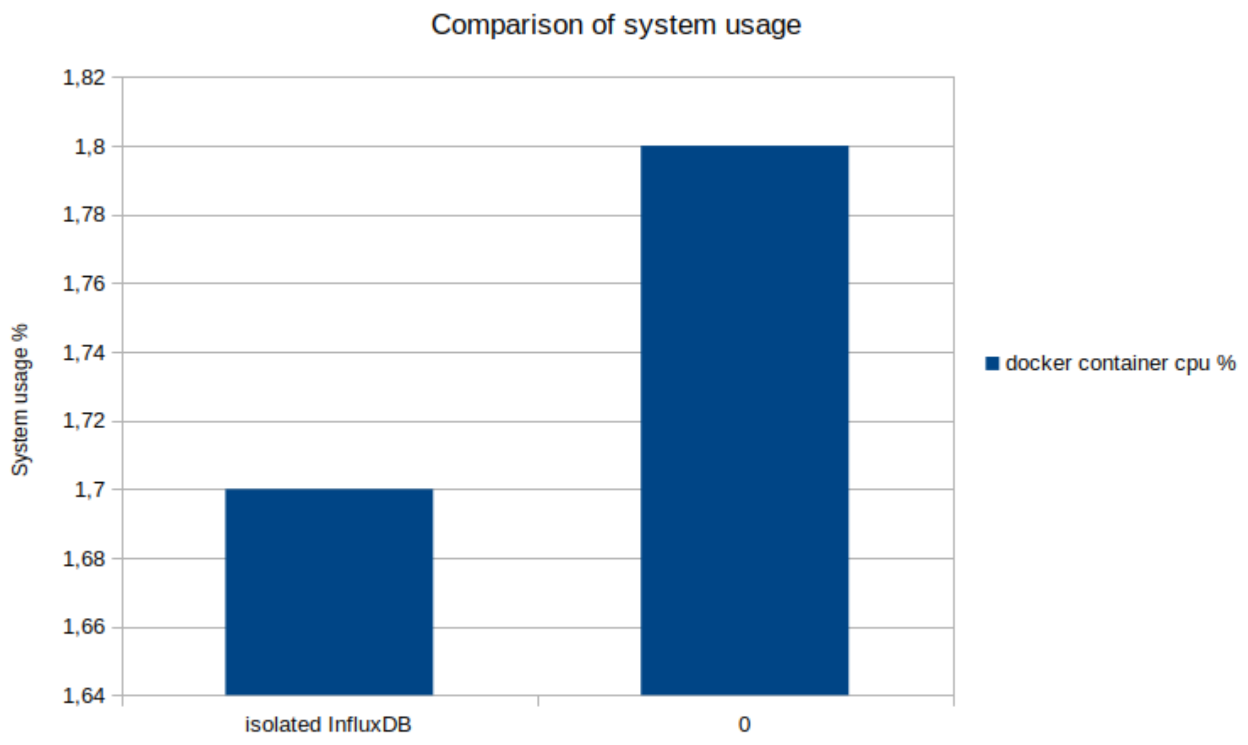

Figure 6.7 system usage comparison: isolated (idle) InfluxDB and InfluxDB gathering data only from the 9 K8s nodes

Given the results we have achieved so far, we can estimate that the Monitoring System can handle a much larger set of monitored applications than the one considered during our tests. However, AI-SPRINT is meant to be used in deployments not larger than 10 nodes hosting a maximum count of 100 jobs (applications) sending data simultaneously to the Monitoring System. Hence, the proposed architecture seems to be sufficient for the given requirements. An evaluation of the KPI K3.2 Monitoring overhead <=5%, will be performed in the AI-SPRINT Deliverable *D3.4 Final release and evaluation of the monitoring system* planned at M24.

7. Improvements and planned development

The technology evaluation and test deployment of the Monitoring Subsystem gave us the possibility to verify the correctness of the architecture we are envisioning. We were also able to check how the Monitoring Subsystem behaves in a real environment and how it can be used by AI-SPRINT applications. We report below the plan for its development and for some improvements. All these features will be available in AI-SPRINT Monitoring Subsystem's final release at M24.

7.1 Deployment

Currently the deployment of the Monitoring Subsystem is not fully automatic. We plan to develop several bash scripts which will help to deploy basic, ready to use Monitoring Subsystem instances and perform typical administrative operations through IM such as adding and removing monitored applications and provide basic dashboards for Grafana. The UPV team will be responsible for creating TOSCA recipes using deployment instructions, configuration files and scripts prepared by our team.

7.2 Client library

The creation of simple proof-of-concept Python applications gave us the opportunity to see which Python libraries are needed by software developers who will create AI-SPRINT Python applications. We plan to extend basic functionalities created for PoC Python applications and create a full-featured Python library which will provide all required features described in Section 2.2.6. This will also require better integration with external alert receiver endpoints which will be developed by other teams for the AI-SPRINT project.

7.3 Hierarchy support

We demonstrated how to successfully provide synchronization data between two separate InfluxDB instances. AI-SPRINT requires that there should be the possibility to provide an automated way to send data from one set of InfluxDB instances to another. We plan to create a few tools which simplify creating hierarchies of InfluxDB databases and data transmission. These mechanisms will be able to be used even when part of the hierarchy will be unavailable for a long period of time.

7.4 Logs gathering and presentation

The final version of the Monitoring Subsystem will support gathering logs from client applications. Logs will be fetched and sent to the ElasticSearch instance. Then they will be able to be searched and displayed in Grafana according to given date-time filters. Also there will be a tool which will allow loading logs generated by applications running on edge devices to the ElasticSearch. Thanks to that feature, system administrators and software developers will be able to debug applications and detect causes of problems.

8. Conclusion

This document summarizes the route taken in order to implement a fully functioning monitoring subsystem able to gather applications and system metrics for AI applications running across computing continua. We reported the experience we gathered in evaluating several technologies available today to develop a monitoring system and we identified the ones we will use as building blocks in the next stages of the AI-SPRINT Monitoring Subsystem development. This deliverable covers the decisions made and future development plans. As we discussed extensively in this document, the main challenges for monitoring applications running in a computing continuum (i.e., scalability, small footprint, low impact on the system, and modular architecture) have been addressed.