PIACERE

DRAFT

Deliverable D6.3

PIACERE run-time monitoring and self-learning, self-healing platform – v3

Editor(s):	Gorka Benguria
Responsible Partner:	TECNALIA
Status-Version:	Draft v1.0
Date:	14.06.2023
Distribution level (CO, PU):	PU

Project Number:	101000162
Project Title:	PIACERE

Title of Deliverable:	PIACERE run-time monitoring and self-learning, self-healing platform –v3
Due Date of Delivery to the EC	31.05.2023

Workpackage responsible for the Deliverable:	WP6 - Monitor plan and self-heal runtime of Infrastructure as Code
Editor(s):	Gorka Benguria, Fundación Tecnalia Research & Innovation
Contributor(s):	Tecnalia (Gorka Benguria, Jesus Lopez, Iñaki Etxaniz), Ericsson (Cosimo Zotti), Polimi (Bin Xiang), XLAB (Ales Cernivec, Tomaz Martincic, Alvaro Garcia Faura), 7bulls (Radosław Piliszek, Marcin Bartmański)
Reviewer(s):	Lorenzo Blasi, HPE
Approved by:	All Partners
Recommended/mandatory readers:	Recommended WP2, WP5, WP7

Abstract:	This deliverable will contain the main outcomes from M25 to M30 of T6.1-T6.4 due to the high dependency of all the different tasks. It will include the monitoring stack coming from task 6.1 with all the time series data collected as well as the monitoring from the security policies from task 6.4, the set of machine learning algorithms (task 6.2) that comprise the self-learning mechanisms and the self-healing strategies (task 6.3) that trigger an optimized redeployment (see WP5). It will be an iterative process. Each deliverable will comprise a Technical Specification Report.

Keyword List:	Monitoring, Forecast, Healing, Security, Availability, Performance

Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/
Disclaimer	This document reflects only the authors' views and neither Agency nor the Commission are responsible for any use that may be made of the information contained therein

DRAFT

Document Description

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	13.10.2023	First TOC and sections assignment	TECNALIA
v0.2	19.10.2023	Comments and suggestions received by consortium partners	TECNALIA
v0.3	16.05.2023	Contributions round 1	7BULLS, ERICSSON, POLIMI, XLAB, TECNALIA
v0.4	18.05.2023	Final Editing	TECNALIA
v0.5	31.05.2023	Intermediate review	HPE
v0.6	02.06.2023	Review addressing	TECNALIA
v0.7	07.06.2023	Final review	HPE
v0.8	09.06.2023	Review addressing	XLAB, TECNALIA
v1.0	14.06.2023	Ready for submission	TECNALIA

DRAFT

BY-SA

Table of contents

Terms and abbreviations 7
Executive Summary 9
1 Introduction 10
1.1 About this deliverable 10
1.2 Document structure 10
2 KR11 Self-learning and self-healing mechanisms overview 11
2.1 Changes in v3 11
2.2 Functional description and requirements coverage 25
2.3 Main Innovations 37
3 KR12 Runtime security monitoring overview 38
3.1 Changes in v3 38
3.2 Functional description and requirements coverage 41
3.3 Main Innovations 45
4 Overview of preliminary experiments 46
5 Lessons learnt and outlook to the future 53
6 Conclusions 55
Annex A. Implementation, delivery and usage 56
A.1. Monitoring Controller 56
A.2. Performance Monitoring 62
A.3. Security Monitoring 68
A.4. Performance Self-learning 73
A.5. Security Self-learning 78
A.6. Self-healing 81

DRAFT

List of tables

Table 1 – Core metrics targeted in v2 11
Table 2 – Core metrics covered in v3 12
Table 3 – Monitoring related user requirements from WP2. 32
Table 4 – Performance Monitoring related internal requirements 35
Table 5 – Security Monitoring requirements from WP2. 44
Table 6 – Security Monitoring related internal requirements. 44
Table 7 – Components and KR relations 56

List of figures

Figure 1 – Performance overview 12
Figure 2 – Performance detail for infrastructure elements 13
Figure 3 – elapsed representation in influxDB 15
Figure 4 – filtering using elapsed in influxDB 15
Figure 5 – Availability detailed view 16

(cc) BY-SA

Figure 6 – IDE access to performance views 17
Figure 7 – PSL logic refactor 18
Figure 8 – Grafana PSL dashboard update 19
Figure 9 – Grafana alerts 19
Figure 10 – Monitoring rule 20
Figure 11 – Grafana deployment alerts 21
Figure 12 – Grafana deployment alerts 21
Figure 13 – Evolution of monitoring agents 22
Figure 14 – Monitoring agents in ICG 23
Figure 15 – Service instance information at IEC 23
Figure 16 – IEC in the DOML concretization. 24
Figure 17 – Performance monitoring internal workflow. 25
Figure 18 – Self-learning workflow diagram. 26
Figure 19 – PSL models stored for each DOML element. 27
Figure 20 – Self-healing internal workflow. 28
Figure 21 – Self-healing sequence diagram. 29
Figure 22 – DOML with default monitoring activated. 30
Figure 23 – DOML with explicit monitoring configuration. 31
Figure 24 – ansible monitoring rule 32
Figure 25 – High-level architecture diagram of Security Monitoring components. 42
Figure 26 – Security Self-learning approach based on LOMOS- anomaly detection workflow from logs. 43
Figure 27 – nginx demo first monitoring integration. 46
Figure 28 – Posidonia demo monitoring integration. 47
Figure 29 – Posidonia performance monitoring agent improvement. 47
Figure 30 – Agents at IEM. 48
Figure 31 – Agents through ICG. 49
Figure 32 – Submodule link to monitoring agents. 49
Figure 33 – SIMPA at IEM. 51
Figure 34 – SIMPA at PRC. 52
Figure 35 – Monitoring Controller internal architecture 57
Figure 36 – Monitoring Controller swagger UI. 61
Figure 37 – Performance Monitoring last version architecture. 62
Figure 38 – Performance Monitoring Controller swagger ui. 66
Figure 39 – Influxdb. 67
Figure 40 – Grafana 67
Figure 41 – Architecture of Security Monitoring and Security Self-learning and the integration between these 68
Figure 42 – High-level internals of LOMOS. 69
Figure 43 – Default page of the Security Monitoring services. 71
Figure 44 – The list of registered agents with the Wazuh's instance 72
Figure 45 – Security Monitoring part of the Security Monitoring Controller API: the upper figure depicts "monitoring" and the lower figure depicts "self_learnin" part of the API. 73
Figure 46 – Architecture of the Self-learning component. 74
Figure 47 – Performance Self-learning OpenAPI. 78
Figure 48 – Self-learning API provided by Security Controller. 80
Figure 49 – Dashboard of the Security Self-learning component: main page. 81
Figure 50 – Dashboard of the Security Self-learning component: inspection of the anomalies detected. 81
Figure 51 – Self-healing internal architecture. 82
Figure 52 – Self-healing project structure. 83
Figure 53 – Self-healing configuration. 85

DRAFT

CC BY-SA

FIGURE 54 – SELF-HEALING PRODUCER. 85
FIGURE 55 – SELF-HEALING CONSUMER. 85
FIGURE 56 – MESSAGES RECEIVED IN THE SELF-HEALING COMPONENT. 87

DRAFT

(cc) BY-SA

Terms and abbreviations

AD	Anomaly Detection
AI	Artificial Intelligence
AMEL	Application Modelling and Execution Language
API	Application Programming Interface
BERT	Bidirectional Encoder Representations from Transformers
CAMEL	Cloud Application Modelling and Execution Language
CSLA	Cloud Service Level Agreements
CSP	Cloud Service Provider
CSV	Comma-separated values
DevOps	Development and Operation
DHCP	Dynamic Host Configuration Protocol
DNN	Deep Neural Networks
DoA	Description of Action
DOML	DevOps Modelling Language
EC	European Commission
ELK	Elastic Logstasth Kibana
EMS	Event Management System
EPA	Event Processing Agents
EPM	Event Processing Manager
EPN	Event Processing Network
FP	False Positive
FT-Tree	Frequent template tree
GA	Grant Agreement to the project
HIDS	Host-based intrusion detection system
HTTPS	Secure HTTP Hyper Text Transport Protocol
HVM	Hypersphere Volume Minimization
IaC	Infrastructure as Code
ICG	Infrastructural Code Generator
IDE	Integrated Development Environment
IDF	Inverse Document Frequency
IDS	Intrusion Detection System
IEC	Infrastructure Elements Catalogue
IEM	IaC execution Manager
IEP	IaC execution Platform
IOP	Infrastructure Optimizer Platform
IPS	Intrusion Prevention System
KR	Key Result
KPI	Key Performance Indicator
LOMOS	LOg MOnitoring System
LSTM	Long Short Term Memory
MAE	Mean Absolute Error
MAPE-K	Monitor-Analyze-Plan-Execute over a shared Knowledge
MCSLAs	Multi-Cloud Service Level Agreements
MLM	Masked Language Modelling

DRAFT

(cc) BY-SA

MSE	Mean Square Error
MTBF	Mean Time Between Failures
MTTR	Mean Time To Recover
NFR	Non-functional Requirements
NSM	Network Security Monitoring
PCA	Principal Component Analysis
PRC	PIACERE Runtime Controller
PSL	Performance Self Learning
QoS	Quality of Service
RCA	Root Cause Analysis
REST	REpresentational "State" Transfer
SEM	Security Event Management
SIEM	Security Information and Event Managements
SLA	Service Level Agreement
SLO	Service Level Objective
SNMP	Simple Network Management Protocol
SOTA	State of the art
SW	Software
TF	Term Frequency
URL	Uniform Resource Locator
UTM	Universal Threat Management
VAST	Visual Analytics Science and Technology
VAT	Vulnerability Assessment Tool

DRAFT

(cc) BY-SA

Executive Summary

This document is a supporting document of the PIACERE run-time monitoring and self-learning, self-healing platform. Therefore, it is one part of the D6.3. The whole D6.3 is composed by:

- The source code of the components that implement the required functionality
- This document

The objective of this document is to present the KRs (Key Results) covered by the WP6, the overview of the preliminary results and the lessons learnt out of this development and experimentation.

The document focusses in the two KRs that build up the PIACERE run-time monitoring and self-learning, self-healing platform. These are:

- **KR11 - Self-learning and self-healing mechanisms** that ensures that the conditions of the Quality of Service (QoS) are met at all times and that a failure or non-compliance of Non Functional Requirements (NFRs) is not likely to occur
- **KR12 - Runtime security monitoring** that verifies any security violation at runtime

For the presentation of each of the KR we present the changes in this last iteration. Following we present the functional description and the requirements coverage. And we finally provide a summary with the main innovations of the KR introduced during this last period.

The functional description provides an overview of the main functionalities of the KR, the main elements that implement them, and the overall flow to accomplish them. The requirements coverage reviews the requirements from the architectural work package (WP2) that are related to the KR.

Additionally, the document includes an appendix (Annex A) for each major component of the **PIACERE run-time monitoring and self-learning, self-healing platform**. For each one we include information about its implementation, delivery and usage. The implementation section contains key information to understand which is the overall internal structure of the components and which technologies have been used to develop them. The delivery and usage sections in the Appendix contain information that will be used during the deployment integration of the WP6 components together with other components from other work packages in the common PIACERE framework.

The current version of the **PIACERE run-time monitoring and self-learning, self-healing platform**, was evolved with three main targets in mind: Finalise the missing features, support the deployment of the components in the use cases, and adapt from the lessons learnt during the application of the KRs in the scenarios.

This version of the document finalises the PIACERE run-time monitoring deliverables series. It extends the previous version with the aspects evolved during the third year of development. Besides it includes a changelog version to understand the evolution during this third year.

DRAFT

1 Introduction

1.1 About this deliverable

This document is a supporting document of the third version (M30) of the PIACERE run-time monitoring and self-learning, self-healing platform. It is a complementary document that explains the approach, the implementation, and the way to deliver and use each one of the current components that take part in the implementation of the functionalities expected from the WP6. Besides, as it is a follow up version of the document, it also covers the evolution with respect to the previous version.

The overall objective in this period has been the finalisation of the latest features and the piloting of the features regarding the performance and security monitoring, self-learning and self-healing components.

This document has been developed merging contributions from all the partners of all the tasks of the WP6:

- Task 6.1 Runtime monitoring and self-healing preparation
- Task 6.2 Self-learning algorithms for failure prediction
- Task 6.3 Strategies and plans for runtime self-healing
- Task 6.4 Runtime security monitoring

The WP6 focusses on the provision of two key results:

- **KR11 - Self-learning and self-healing mechanisms** that ensures that the conditions of the QoS are met at all times and that a failure or non-compliance of NFRs is not likely to occur.
- **KR12 – Runtime security monitoring** that verifies any security violation at runtime

—The purpose of this document is threefold:

- To serve as a reference of the background of the technical decisions taken regarding the approaches followed during the development of the components
- To contain information to support future development. This includes information to understand how the components have been developed, which are their features and how can be tested.
- To describe the evolution with respect to the previous version.

DRAFT

1.2 Document structure

The document is structured into six parts. Section 2, focussed on the KR11, presents the final evolution, the functions covered and their main innovations. Section 3, focussed on the KR12, in the same way presents the final evolution the functions covered and their main innovations.

Following the section 4 presents the overview of the preliminary results in the application of the components in the WP6 scenarios. In the next section 5 we summarize the lessons learnt and the outlook to the future.

Finally, we present the conclusions of the deliverable D6.3.

Besides, there is a final part Annex A that addresses the implementation, delivery and usage of each component. The implementation the delivery and usage parts, have been designed to be used in isolation by the developers, without requiring them to read the whole document. With that objective in mind some figures may be repeated to improve that isolated readability.

CC BY-SA

2 KR11 Self-learning and self-healing mechanisms overview

KR11 is related with T6.1, T6.2, and T6.3.

KR11 – Self-learning and self-healing mechanisms ensures that the conditions of the QoS are met at all times and that a failure or non-compliance of NFRs is not likely to occur.

KR11 provides mechanisms that allow to seamlessly embed monitoring mechanisms over the infrastructure that run the applications. This monitoring mechanisms continuously inspect some critical aspects that indicate situations in which the underlying infrastructure may fail to support the application. Besides, applies Artificial Intelligence (IA). techniques to allow some prediction capabilities for being able to foresee these situations with some time to be able to perform some mitigation actions before the application is affected and thus the users. In that sense it provides some self-healing infrastructure that allows to apply short term strategies to minimize potential consequences of these infrastructure degradations.

2.1 Changes in v3

During this last period, we have mainly worked in the finalization and application of the KR11 on the scenarios. Besides, some adjustments have been performed based on the lessons learnt during the usage of the KR11 in the scenarios. In summary, the changes introduced in this last period are:

- Metrics have been simplified
- Availability has been introduced
- DOML has been extended
- Self-healing strategies have been extended
- Monitoring agents have been integrated with ICG
- Self-healing reports alerts to the catalogue

2.1.1 Core metrics simplification

In the previous version we were focussing the self-learning and self-healing activities on eight core metrics: three for CPU, two for memory, two for disk and an additional one, which is availability, as shown in the following Table 1.

Table 1 – Core metrics targeted in v2

CPU	Disk	Mem	availability
Idle percent System percent User percent	Free Used percent	Free Used percent	availability

These metrics were using different scales, for example free disk was measured in gigabytes while CPU idle was percentual. Besides, even in the same category they should be interpreted in different ways: idle percent is better the higher it is, while system and user percent are better as they get lower. Finally, from an event identification perspective they were redundant. For example, if we have low free memory, we are going to have high memory used percent, therefore there was no point to track both in a default scenario. All this divergence and the introduction of additional metrics were causing some problems:

- Add unnecessary complexity from a use case point of view.
- Require non necessary work in the backend, each extra metric requires additional development, maintenance and processing.

DRAFT

(cc) BY-SA

- Require extra storage in the infrastructure

As a conclusion we have established a baseline configuration with four key metrics: CPU, Disk, Mem and availability as shown in Table 2.

Table 2 – Core metrics covered in v3

CPU	Disk	Mem	availability
Used percent	Used percent	Used percent	Availability percent

All of them are percentual so they share the same scale: this will facilitate the metrics understanding from the use case perspective in the default scenario. Besides, most of them are interpreted in the same way: the higher they are the worst they are. The exception for this rule is availability that is worst the lower it is. In the Figure 1 we show the performance overview where we can see the metrics and the difference in the interpretation of the system availability.

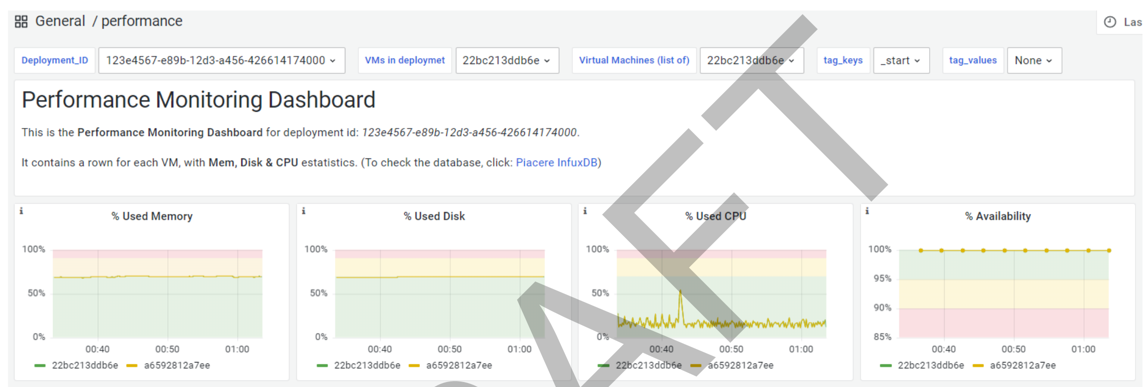


Figure 1 – Performance overview

It is worth mentioning that, even if we are going to focus our default algorithms and processing on this four metrics, we still collect the metrics used before, and we have added additional ones that are useful to interpret the behaviour of the infrastructure elements as shown in Figure 2. We will refer to them as supporting metrics.

DRAFT

(cc) BY-SA

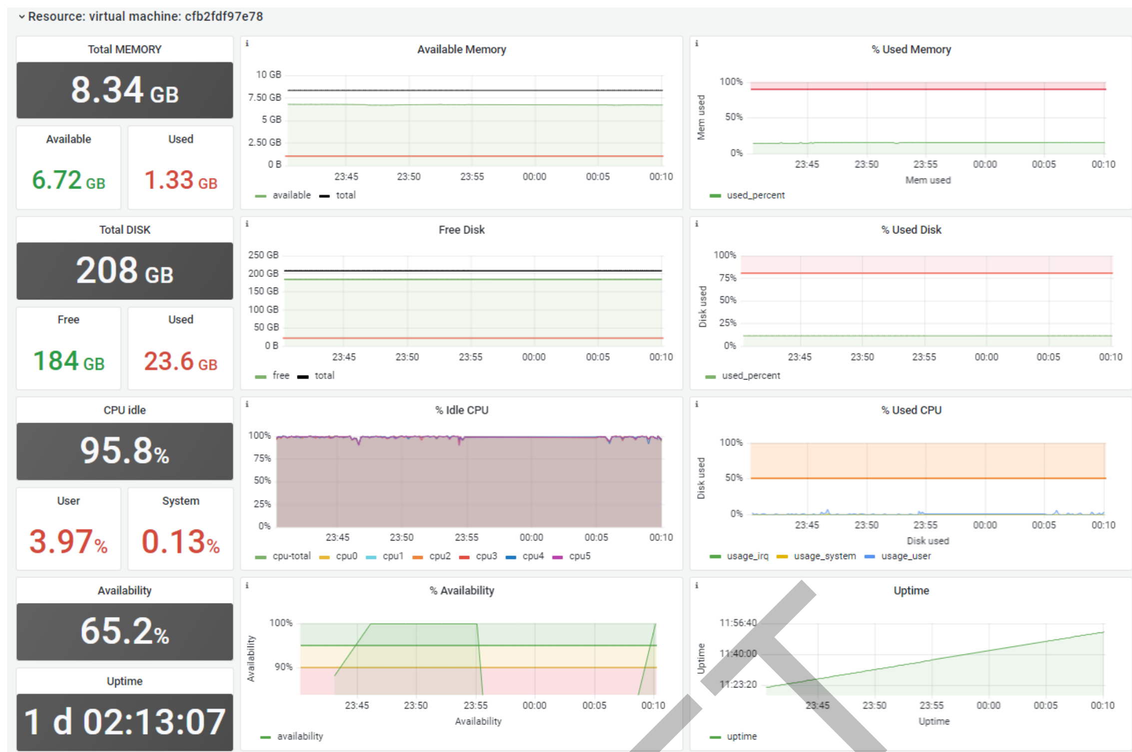


Figure 2 – Performance detail for infrastructure elements

Implementing these four metrics has been challenging as two of them, CPU and availability, were not directly supported from the monitoring infrastructure chosen for the KR11. Telegraf provides many metrics relative to the CPU, but there is no metric that reports directly the CPU used percent. To calculate it, we compute the complement of the usage_idle.

```
cpu = deploymentData
  |> filter(fn: (r) => r["_measurement"] == "cpu")
  |> filter(fn: (r) => r["_field"] == "usage_idle")
  |> filter(fn: (r) => r["cpu"] == "cpu-total")
  |> map(fn: (r) => ({ r with
      _field: r._measurement + "_" + "used_percent"
  }))
  |> map(fn: (r) => ({ r with
      _value:  100.0 - float(v: r._value)
  }))
  |> keep(columns: ["_time", "_field","_value","doml_element_name", "host"])
```

This flux code snippet shows how the CPU used percent is calculated as the complement of the usage idle percentage. DeploymentData is a range of time series data (for example metrics during the last 30 min) for the deployment id under analysis. This data range contains all the metrics CPU, memory, disk, system, etc. Therefore, the next step is to focus only in the metric and field of interest, in this case CPU and usage idle. Then we use the map flux function to redefine the field name to “cpu_used_percent” and the value to the complement of usage_idle as we stated before.

This code snippet is used in flux queries in different components along the PIACERE run-time monitoring and self-learning, self-healing platform. It is used in Grafana for populating the dashboard and configure alerts. It is used in Performance Self-Learning (PSL) to gather the information used for training and prediction. Regarding the computation of the availability, this is summarized in an upcoming section.

Another action that has been introduced for the sake of simplification has been the introduction of default thresholds. We have established for most of the metrics a warning threshold of the

DRAFT

CC BY-SA

70% and a critical threshold of 90%. This has been applied to the three first metrics: memory, disk and CPU. The exception again has been availability that due to its nature has different default thresholds.

2.1.2 Availability

The availability metric has been added to check the readiness of the different DOML elements in the infrastructure. The measurement of the availability of an infrastructure element is a challenging activity as it usually cannot be reported from the involved element. In fact, there is no direct support for this in the Telegraf package. Telegraf is the package that has been chosen for the metrics collection. We have evaluated two strategies for gathering the availability:

- External strategy: The external strategy implies the configuration of an additional monitoring component that will "ping" the infrastructure. This has a couple of drawbacks. First it requires to expose a service on the network to be accessible to the monitoring component. This obviously, implies a security risk. Second, the monitored component must be accessible from the monitoring component: this may create problems in complex networking scenarios. For example, in a perimetral network security approach some components are not accessible unless the monitoring component is inside the network. Another example are those cloud service providers that constrain the number of public IP addresses available, forcing the use of internal networks for some of the components of the infrastructure.
- Internal strategy: The internal strategy implies the usage of the metrics that we have, to derive the availability. In this sense we can exploit the concept of gaps. To compute the gaps we will exploit the *elapsed()* InfluxDB flux function[1], which provides information about the time elapsed between two records of the same metric.

We have chosen the internal strategy, as in cloud deployments the direct access to all the monitorable infrastructure elements is not always guaranteed. Up to this period we have applied monitoring in different kinds of infrastructure elements such as virtual machines and containers with different operating systems.

The availability calculation makes use of the *elapsed()* flux function and the fact that usually the time between metrics matches the Telegraf interval.

```
sourceData = from(bucket: "bucket")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "system")
  |> filter(fn: (r) => r["_field"] == "uptime")
  |> filter(fn: (r) => r["deployment_id"] == "123e4567-e89b-12d3-a456-426614174000")
  |> filter(fn: (r) => r["host"] == "35069d0c3311")
  |> keep ( columns: ["_time", "_measurement", "_field", "deployment_id", "_value",
"host"])

betweenRecordTime = sourceData
  |> elapsed()
```

This flux code snippet is used to represent how the *elapsed* function is used within the flux language.

[1] https://docs.influxdata.com/flux/v0.x/stdlib/universe/elapsed/,

DRAFT
(cc) BY-SA

false	true	true	true	true	false
long	string	string	string	string	long
_value	_field	_measurement	deployment_id	host	elapsed
3249	uptime	system	123e4567-e89b-12d…	35069d0c3311	10
3259	uptime	system	123e4567-e89b-12d…	35069d0c3311	10
3269	uptime	system	123e4567-e89b-12d…	35069d0c3311	10
3279	uptime	system	123e4567-e89b-12d…	35069d0c3311	10
3289	uptime	system	123e4567-e89b-12d…	35069d0c3311	10

Figure 3 – elapsed representation in influxDB

Figure 3 represents the *elapsed* function effect in flux language, there we can see the additional column that appears when applying the *elapsed* function in the flux language. The column contains the number of seconds elapsed between values of the same metric and element.

This can be used to compute the past availability. If values over 10 that is the established TELEGRAF_INTERVAL, we can add a filter to get only the values greater than 10, for example.

```
sourceData = from(bucket: "bucket")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "system")
  |> filter(fn: (r) => r["_field"] == "uptime")
  |> filter(fn: (r) => r["deployment_id"] == "123e4567-e89b-12d3-a456-426614174000")
  |> filter(fn: (r) => r["host"] == "35069d0c3311")
  |> keep ( columns: ["_time", "_measurement", "_field", "deployment_id", "_value",
"host"])

betweenRecordTime = sourceData
  |> elapsed()
  |> filter(fn: (r) => r.elapsed > 15)
```

This flux code snippet is used to represent how the *elapsed* function can be used to perform additional transformations over the metrics sets obtained from influxDB. The visual result of this filtering is shown in Figure 4.

false	true	true	true	true	false
long	string	string	string	string	long
_value	_field	_measurement	deployment_id	host	elapsed
5829	uptime	system	123e4567-e89b-12d…	35069d0c3311	750

Figure 4 – filtering using elapsed in influxDB

This approach is useful to calculate gaps inside a stream of metrics, but it is not capable to calculate the gaps in the edges of the range. Elapsed flux function calculates the time between consecutive metrics therefore if there is no metrics, then the elapsed function is not useful. To solve this difficulty, we used the fill flux function to assume the lack of elapsed data as gaps.

```
bucket = "bucket"
start = -30m
deploymentId = "123e4567-e89b-12d3-a456-426614174000"
telegrafInterval = 10s
discardLevel = int(v: duration(v: uint(v: telegrafInterval) + uint(v: 5s)))/1000000000

deploymentData = from(bucket: bucket )
|> range(start: start )
|> filter(fn: (r) => r["deployment_id"] == deploymentId )

availability = deploymentData
|> filter(fn: (r) => r["_measurement"] == "system")
|> filter(fn: (r) => r["_field"] == "uptime")
|> elapsed()
|> map(fn: (r) => ({ r with
  _field: r._measurement + "_" + "availability_percent"
```

DRAFT

(cc) BY-SA

```
}))
|> map(fn: (r) => ({ r with
  _value: if r.elapsed > discardLevel then 0 else 100
}))
|> keep(columns: ["_time", "_field","_value","host"])
|> aggregateWindow(every: telegrafInterval, fn: mean, createEmpty: true)
|> fill(column: "_value", value: 0.0)
|> tail(n: 1000000,offset: 1)
|> keep(columns: ["_time", "_field", "_value", "host"])
```

In the upper segment of flux code, we perform the following steps:

- We request a range of metrics for a given deployment, and we store in deploymentData, this will include a stream with all the collected metrics for all the infrastructure elements that are labelled with the deployment id.
- Next, we focus on a single metric to reduce the amount of data, for that purpose we use uptime. We have introduced uptime in this release as we have seen it as interesting complement for the availability metric. In case an availability issue, if the uptime suffers a reset this will inform us about a possible reset. In case the uptime is not reset this may be produced by network issue or a system freeze.
- Next, we evaluate the elapsed time between metrics, if we have elapsed time and it is lower that the TELEGRAF_INTERVAL with some margin we assume it as an available micro time period and we assign it 100% value. In case the value is greater we assume as a availability gap and we assign 0%.
- Next we aggregate each Telegraf interval to have metrics every TELEGRAF_INTERVAL: even if in reality we haven't received nothing in those periods, it creates metrics with empty values.
- Next, we force to 0% the empty values using the fill function
- Finally we remove the last availability metric, because it may not match exactly a metric function, which leads to means with few seconds (< TELEGRAF_INTERVAL) of assumed 0 values, that are based on assumptions rather than on real values.

This provide us with a stream of 0%/100% every TELEGRAF_INTERVAL that we can latter aggregate in more significant time periods to get an estimation of the availability for those periods based on the assumption that the lack of Telegraf metrics flow from a monitored infrastructure element is a sign of potential availability problem.

Besides, apart from the performance overview view (Figure 1), we have added a detailed availability view, with details for specific elements in the infrastructure. It can be seen in the next Figure 5.

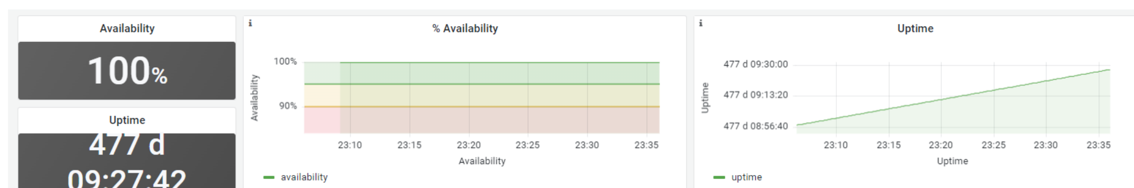


Figure 5 – Availability detailed view

Both views are provided in the Grafana component of the monitoring element. These views are accessible as before from the IDE component for each deployment.

DRAFT

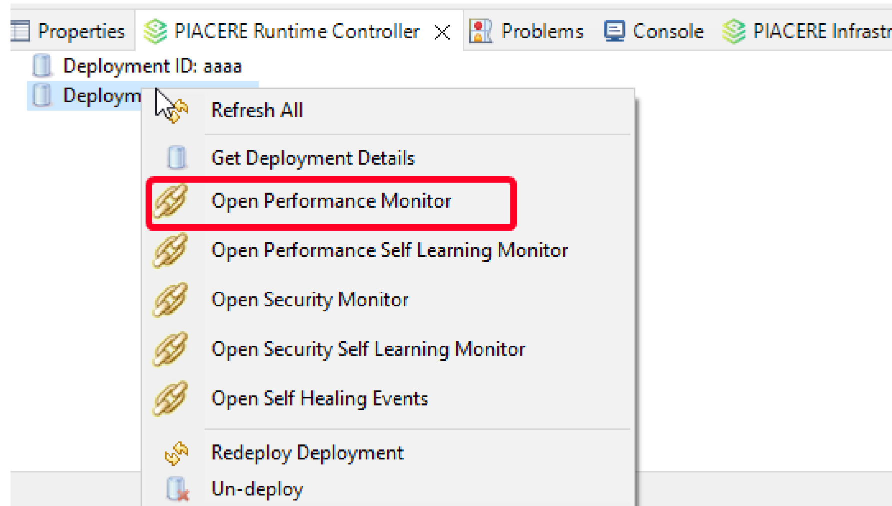


Figure 6 – IDE access to performance views

The detailed availability view (Figure 5) provides four information elements:

- The availability percentage for the monitored range for a concrete element, this gives us the possibility to calculate the percentages for a concrete day, hour, etc.
- The current total uptime, this may be useful for the diagnosis of availability problems
- The availability evolution for a concrete element in the monitored range.
- The uptime evolution for a concrete element in the monitored range.

2.1.3 Self-Learning

The self-learning component has evolved in several aspects:

- Fixes in stability
- Focus on four main metrics
- Increase granularity
- Update Grafana dashboard
- Delegate notification to Grafana

The stability improvement has come from the application of the self-learning module to a long-term living deployment. We found out that after some hours or days the self-learning module was stopping suddenly. The reason for that was that in some cases the InfluxDB fails to provide metrics, which drives to a stop in the iterative process. We fixed the problem, allowing to skip calculations in one iteration if influxDB fails to provide data.

The next change was the corresponding change related with the simplification of metrics already reported in section 2.1.1. Apart from the removal of metrics, this also implied the implementation and testing of self-learning processing for the new metrics introduced:

- CPU used percent
- System availability percent

We have also introduced a major change regarding the granularity of the metric analysis. In previous releases we focussed on the deployment as a whole. In this release we have increased the granularity to the different DOML elements in the infrastructure. This impacted on the different parts of the algorithm:

DRAFT

- We have changed the information request to return information on all the DOML elements.
- We have introduced a cycle in the deployment processing to perform training and predictions for each of the DOML elements inside each deployment.
- We save data in the InfluxDB for each DOML element in the deployment.

In the following Figure 7 we show the changes introduced in the main flow for the increase in the granularity. We iterate each considered deployment (1) and each deployment has its own model set (2). For each deployment we iterate its DOML elements (3) and we generate their own training models (4) that are stored for their latter use and update by the predictions.

```python
def run():
    logging.info("Running Self Learning Flow")
    start_time = time.time()
    deployments = deployments_repository.get_all()
    total_deployments = len(deployments)
    valid_predictions = 0
    for deployment_id in deployments.keys():
        deployment_id = deployment_id.decode(ENCODING)
        logging.info("Processing Deployment: {}".format(deployment_id))
        latest_metrics = metric_source.get_latest_metrics(deployment_id)

        # group latest metrics by doml_element_name
        if latest_metrics is None:
            logging.info("No metrics found for deployment: {}".format(deployment_id))
            continue
        latest_metrics_groupby_doml_element_name = latest_metrics.groupby("doml_element_name")
        # iterate over each doml_element_name
        for doml_element_name, doml_element_name_metrics in latest_metrics_groupby_doml_element_name:
            # check if doml_element_name has training model
            predictor.train_deployment_id_doml_element_name(deployment_id, doml_element_name)
            prediction = None
            with warnings.catch_warnings():
                # ignore all caught warnings
                warnings.filterwarnings("ignore")
                prediction = predictor.predict_deployment_id(deployment_id, doml_element_name,
                latest_metrics=doml_element_name_metrics)
            is_valid_prediction = False
            try:
                prediction_datetime = pandas.to_datetime(prediction[-2]).replace(tzinfo=None)
                input_datetime = pandas.to_datetime(prediction[-1]).replace(tzinfo=None)
                is_valid_prediction = True
                valid_predictions += 1
            except Exception:
                logging.info("Prediction for deployment {} is invalid".format(deployment_id))
            if is_valid_prediction:
                logging.info("Prediction for deployment {} is {} note that forecast time is given at
                GMT".format(deployment_id, prediction))
                influxdb.save_prediction(prediction, lab_names, prediction_datetime, deployment_id,
                doml_element_name, input_datetime)
                logging.info("Saved at: {}".format(input_datetime))
                check_thresholds(deployment_id, doml_element_name, prediction)
    total_time = time.time() - start_time
    logging.info("Self Learning Flow: Valid predictions {}, total deployments {}, time {} seconds".
    format(valid_predictions, total_deployments, total_time))
    return valid_predictions, total_deployments, total_time
```

Figure 7 – PSL logic refactor

Grafana dashboard focused on the self-learning metrics, has been updated with the new metrics, and the non-necessary ones have been removed, as shown in Figure 8.

CC BY-SA

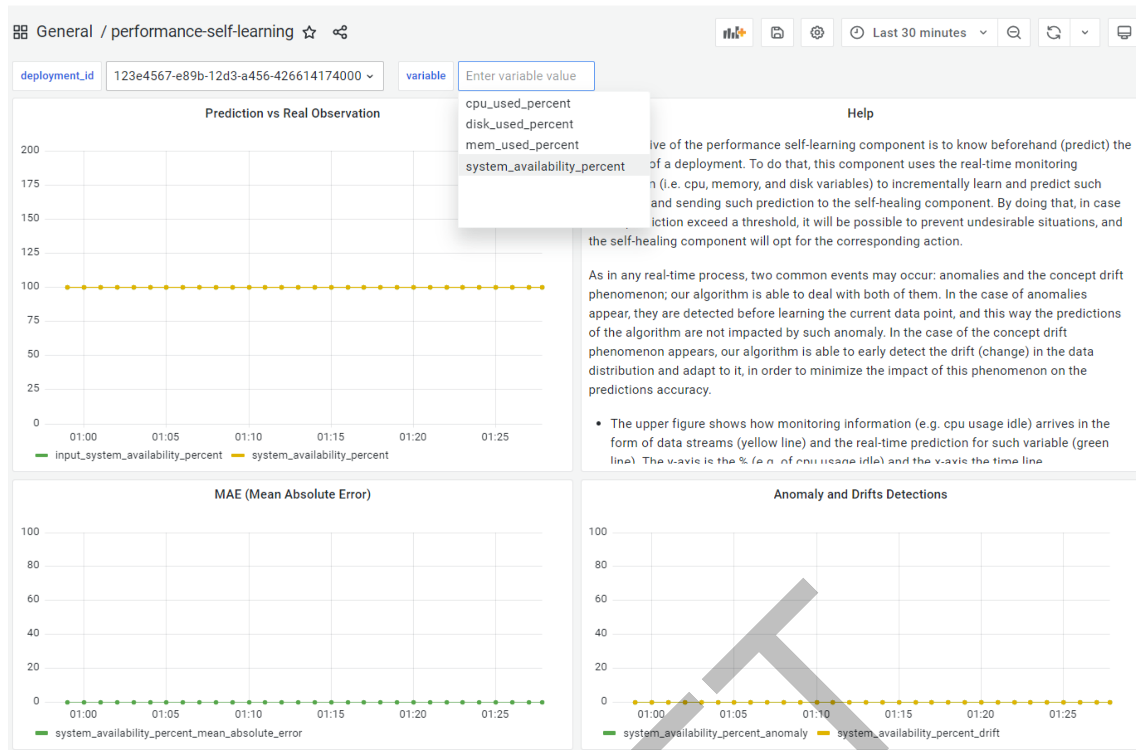


Figure 8 – Grafana PSL dashboard update

DRAFT

Finally, we have introduced a change in the notification approach where we centralize both monitoring and alerting by exploiting the Grafana capabilities, as shown in Figure 9.

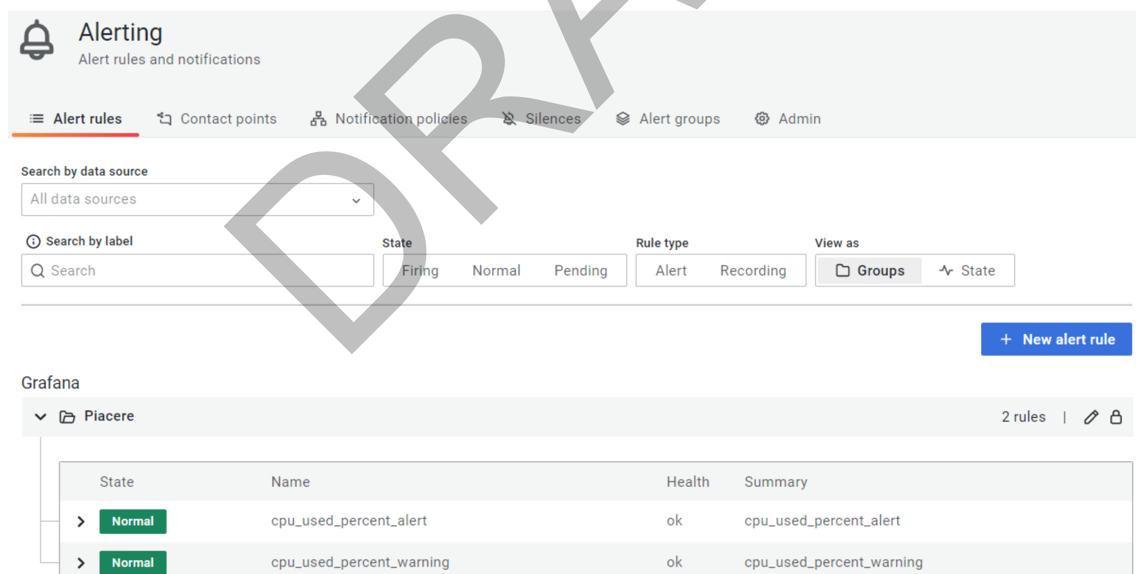


Figure 9 – Grafana alerts

This allows more flexibility and the exploitation of the DOML extensions that will be explained next.

2.1.4 DOML

Another change in this iteration has been the extension of DOML to support both KR11 and KR12 in two aspects:

CC BY-SA

- Allow the user to disable monitoring for some elements
- Enable monitoring rules

One need was to have the possibility of disabling the performance or security monitoring for specific elements of the infrastructure. To support this need DOML has been extended with the capability of specifying a list of monitoring elements to be disabled.

```
infrastructure infra {
      vm igw_vm {
            os "Ubuntu-Focal-20.04-Daily-2022-04-19"
            size "small"

            iface igw_vm_oam {
                  belongs_to subnet_oam_igw
            }

            iface igw_vm_net1 {
                  belongs_to subnet_net1_igw
            }

            credentials ssh_key

            disabled_monitorings "performance,security"
      }
```

The upper DOML code snippet exemplifies how to disable monitoring features. Currently, we implement two monitoring features: performance (which includes availability) and security.

To disable monitoring a "disabled_monitorings" element must be added indicating, inside a comma separated list, the monitoring features to be disabled. This is latter processed by the ICG that will add or skip the inclusion of the corresponding monitoring agents of that element based on this information.

Another addition was the "monitoring_rules". The objective is to extend the monitoring and self-healing capabilities.

```
monitoring_rule disk_usage_over_50 {
    /*
     * A formal string attribute, whose value is dependent on the strategy attribute,
     * that defines the condition that will trigger the monitoring.
     */
    cond "disk_used_percent > 50"
    // A string attribute to specify the name of the monitoring strategy this rule will use.
    strat "notify"
    // An optional string attribute to specify parameters valid for the given monitoring strategy.
    config "Disk usage over 50%"
}
```

Figure 10 – Monitoring rule

From the monitoring perspective we can establish additional monitoring rules (see an example in Figure 10) that will add additional monitoring alerts at Grafana (Figure 11).

DRAFT

CC BY-SA

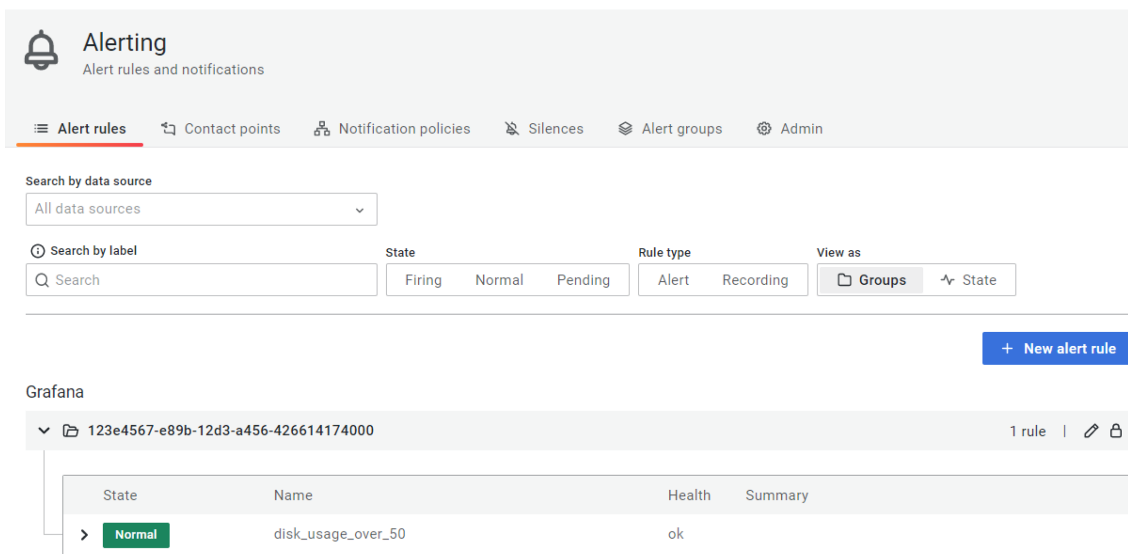


Figure 11 – Grafana deployment alerts

2.1.5 Self-healing

The self-healing component has been modified in the event reception and in the strategies execution. Regarding the event reception a new channel has been added that enables the receptions of web-hooks from Grafana. The channel added is shown in the Figure 11.

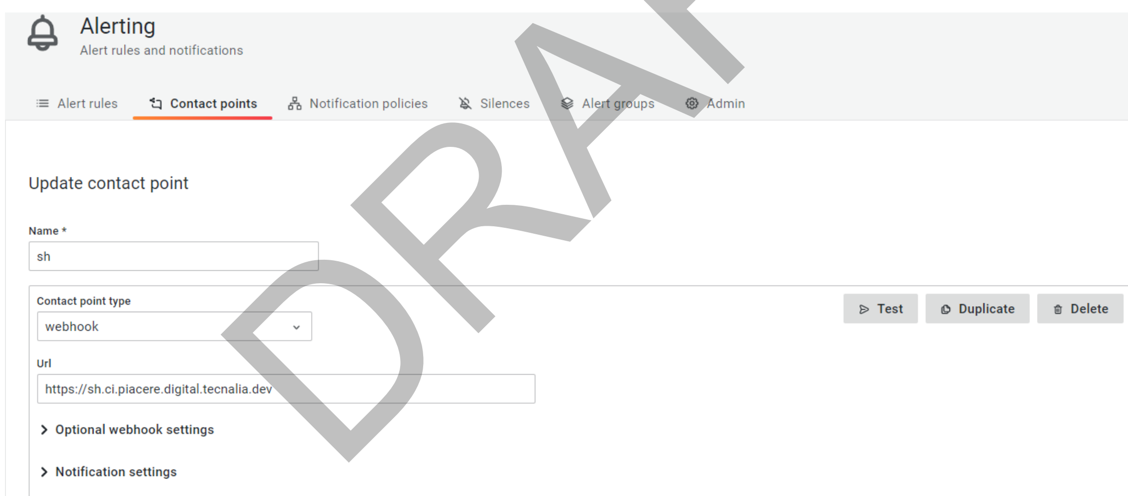


Figure 12 – Grafana deployment alerts

Regarding the strategies execution, it has been modified to receive and process the monitoring rules from the DOML. Monitoring rules as shown in Figure 10, apart from the condition that is used to trigger the webhook, contain information about the strategy to apply and the configuration for that strategy. The strategies currently contemplated are: notify, restart, ansible and scale. The configuration depends on the strategy, ansible strategy being the more complex.

- *Notify* requires a message
- *Restart* does not require configuration
- *scale* requires information about how the scale will be apply
- *Ansible* requires identifying the source of the playbook and its attributes

Note: additional details on the strategies are provided in the following section 2.2

CC BY-SA
DRAFT

Except for notify strategy, the other strategies require communication with the PRC in order to interact with the IEM in a coordinated way.

- *Restart* is a preconfigured ansible strategy with reboot playbook
- *Scale* requests a redeploy changing the DOML regenerating the IaC and executing Ansible if present.
- *Ansible* strategy executes an ansible playbook over the inventory of a preexisting deployment.

2.1.6 Monitoring Agents

Another change area has been the monitoring agents. During this last iteration monitoring agents have advanced in multiplatform support and in the integration with the ICG. Performance monitoring agents have been proven to be multiplatform as they have been installed in Debian and fedora systems deployed in OpenStack, vSphere datacenters and docker based containers.

Besides, the performance monitoring agent has been modified to:

- Include additional labels to support the correlation of deployed elements with DOML elements in the infrastructure layer.
- Include new supporting metrics to support the understanding of availability issues.

DRAFT

```yaml
pma_deployment_id: "{{ lookup('env', 'DEPLOYMENT_ID' ) }}"
pma_influxdb_bucket:  "{{ lookup('env', 'INFLUXDB_BUCKET' ) }}"
pma_influxdb_token:  "{{ lookup('env', 'INFLUXDB_TOKEN' ) }}"
pma_influxdb_org:  "{{ lookup('env', 'INFLUXDB_ORG' ) }}"
pma_influxdb_addr:  "{{ lookup('env', 'INFLUXDB_ADDR' ) }}"

telegraf_agent_package_state: latest

telegraf_agent_output:
  - type: influxdb_v2
    config:
      - urls = ["{{ pma_influxdb_addr }}"]
      - token = "{{ pma_influxdb_token }}"
      - organization = "{{ pma_influxdb_org }}"
      - bucket = "{{ pma_influxdb_bucket }}"
      - insecure_skip_verify = true

telegraf_global_tags:
  - tag_name: deployment_id
    tag_value: "{{ pma_deployment_id }}"
  - tag_name: doml_element_name
    tag_value: "{{ doml_element_name }}"
  - tag_name: doml_element_type
    tag_value: "vm"

telegraf_plugins_default:
  - plugin: cpu
  - plugin: mem
  - plugin: processes
  - plugin: disk
  - plugin: net
  - plugin: system
```

Figure 13 – Evolution of monitoring agents

(cc) BY-SA

The Figure 13 shows the changes that we introduced in the agents configuration presented above. (1) Additional label was added to allow the focusing of the monitoring and the self-healing over concrete DOML elements. In previous releases we were using hostname, but this was not useful to correlate the monitored metrics with concrete DOML elements. (2) Additional support metrics were activated to facilitate the interpretation of monitoring results.

Regarding the integration of the monitoring agents in the ICG, we switch to a git submodule-based approach (Figure 14), with facilitates the agile development and evolution of the monitoring agents while the ICG evolves. Monitoring agent submodules contain IaC code that installs, configures, and starts the monitoring agents in the created infrastructure. This IaC code is included in the IaC code generated by the ICG.

y3 t34-icg-controller / templates / ansible / cross-platform / + History Find file Web IDE Clone

Name	Last commit	Last update
..		
performance_monitoring @ 6321619b	updates performance monitoring with uptime	4 days ago
security_monitoring @ 8b06981d	refactors monitoring agents	2 months ago

Figure 14 – Monitoring agents in ICG

2.1.1 Catalogue feedback

The self-healing component feeds the Infrastructure Element Catalogue (IEC) with information about the alerts in the used services. Each time an event is received it is registered in the IEC as an alert (Figure 15).

Home Services Service Classes Instances Images Existing Resources Administration

nginx_host N.Alerts 0

Deployment ID 123e4567-e89b-12d3-a456-426614174000
Name nginx_host
Instance Ip Url 172.26.126.121
Email testuser@tecnalia.com
Service m1.medium

Alerts:

Back

Figure 15 – Service instance information at IEC

To do so, each time an event is received for a DOML element in the infrastructure layer we need to identify the service in the IEC which it is related to it and upload the event to that service.

DRAFT

(cc) BY-SA

```
doml uc3_openstack

application app {

    // need to define all sw components of the project this is a placeholder
    // need to understand what is really needed in this spec
    // need to specify all provides/consumes

    software_component iwg {
        provides { net_info }
    }
    software_component osint {
        provides { osint_info }
        }
    }

infrastructure abstract_infra {
    vm absA2v2_USA {
        mem_mb 4096.0
        sto "20480.0"
        cpu_count 2
        cost 67.0
    }
}
concretizations {
    concrete_infrastructure concrete_infra {
        provider azure {
            vm A2v2_USA
            properties {
                ^availability = 95.0
                ^performance = 5.0
            }
            {
                maps absA2v2_USA
            }
        }
    }
}
```

Figure 16 – IEC in the DOML concretization.

Figure 16 shows how the IDE inserts elements from the IEC in the DOML. We use that structure to correlate DOML elements with IEC services. When a service is added (1) a *provider* element is created inside the concrete infrastructure, and inside that provider a *vm* element is added with the name of the IEC service used. This is the name that we have to correlate with the DOML element. Within the *vm* in the concretization layer, a *maps* element is added (2); this *maps* element contains the DOML element name of the element in the infrastructure layer it maps to (3).

2.1.2 Monitoring controller

Finally, monitoring controller, that is a utility component shared between KR11 and KR12, has been updated to forward the deployment bundle to monitoring components and the self-healing component.

- Monitoring components require the DOML specification included in the bundle to extend the alerts in the Grafana component based on the conditions of the monitoring rule. Alerts are configured based on the conditions and forward the strategy and configuration through the webhook.
- Self-healing component requires the bundle in order to include changes based on the strategies and configuration received. Modified bundle is then sent back to the PRC for processing.

CC BY-SA

2.2 Functional description and requirements coverage

The KR11 provides functionalities focused on enabling and supporting self-learning and self-healing mechanisms. To implement this objective, it implements three main functionalities:

- Basic performance monitoring
- Self-learning
- Self-healing

The first functionality is the **basic performance and availability monitoring** this is crucial to support the self-learning and the self-healing. On one hand, the self-learning requires a metric stream in order to be able to learn and provide insights. On the other hand, self-healing needs to receive events when metrics require implementation of some mitigation actions.

The following Figure 17 represents the internal workflow of the performance monitoring components and their internal parts. Based on the request from the user, typically done from the IDE, the PRC (1) requests the activation of the deployment to the IEM (2) and to the monitoring stack (3).

The IEM deploys the IaC generated by the ICG, that contains the monitoring agents (4). This will create somewhere the infrastructure elements required by the application, and these infrastructure elements will contain the monitoring agents. These monitoring agents will provide basic information periodically that will be stored in a time series database (5). The time series database selected for the project is InfluxDB. We include it as a containerized image, and it is secured with a basic password protection. The database access details are parametrized in all the elements that use it, so that we can replace it for a more capable database in case it is necessary. In this sense InfluxDB, is deployable in many ways and it can even be contracted as a SaaS.

The monitoring stack receives the deployment activation request from the PRC in the monitoring controller (3). It forwards this request to lower-level monitoring components such as performance monitoring controller (6). The performance monitoring controller configures the appropriate dashboards and alerts (7). The dashboard will take care of the visualization of metrics with different purposes, the alerts will take care of the communication of events to the self-healing component.

DRAFT

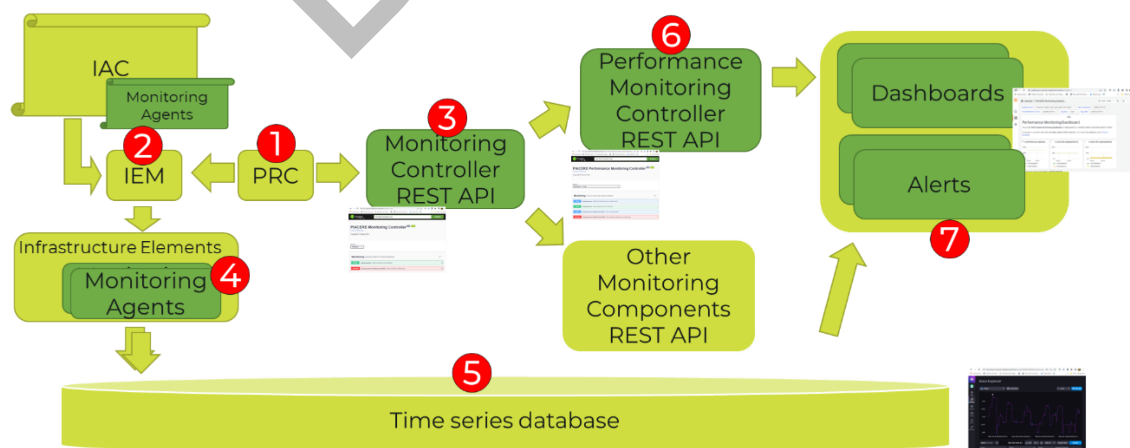


Figure 17 – Performance monitoring internal workflow.

(cc) BY-SA

The second functionality is the performance **self-learning**. The objective of the self-learning component is to observe the metrics that are continuously gathered and find insights that improve the self-healing capabilities of the overall system.

The following Figure 18 describes the internal workflow in the performance self-learning. The self-learning iteratively processes metrics (1). It takes the list of tracked deployments (2), which are loaded by the monitoring controller under request from the PRC.

For each active deployment the self-learning issues a query to the time series database (3) to retrieve the latest metrics for all the infrastructure elements active in the last period. This can potentially return metric values for several DOML elements: those DOML elements that are defined in the DOML specification and have been available during the last half hour.

The processing starts with checking if the training has been done (4); if the training is not done, (5) it checks if enough metrics are in place to create the initial models (training requires a minimum of 200 metric values by default, but this is configurable). Each covered metric will have its own model for each DOML model element of each deployment id (Figure 19). In this last version four metrics are considered as presented before: CPU, memory, disk, and availability.

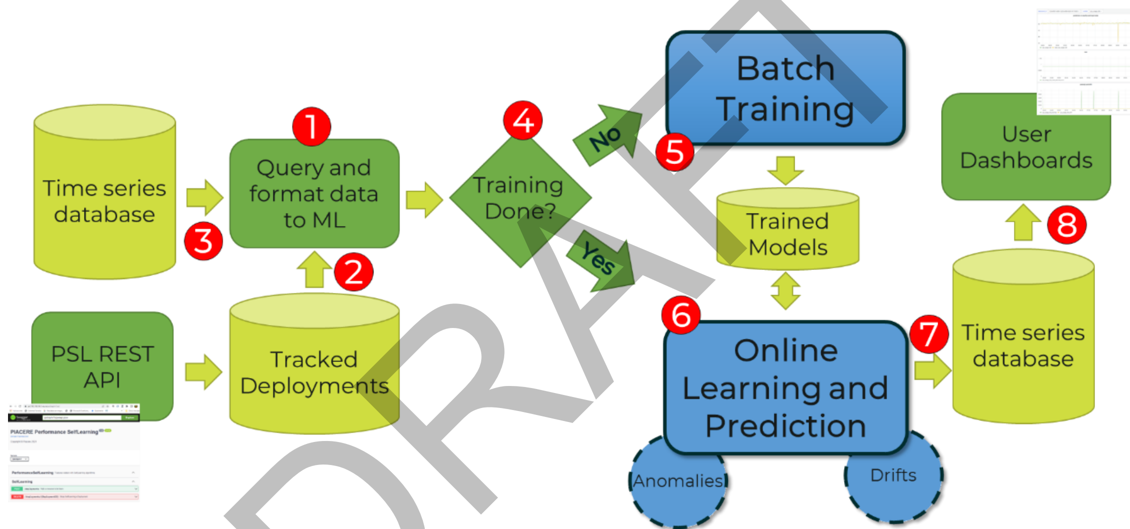
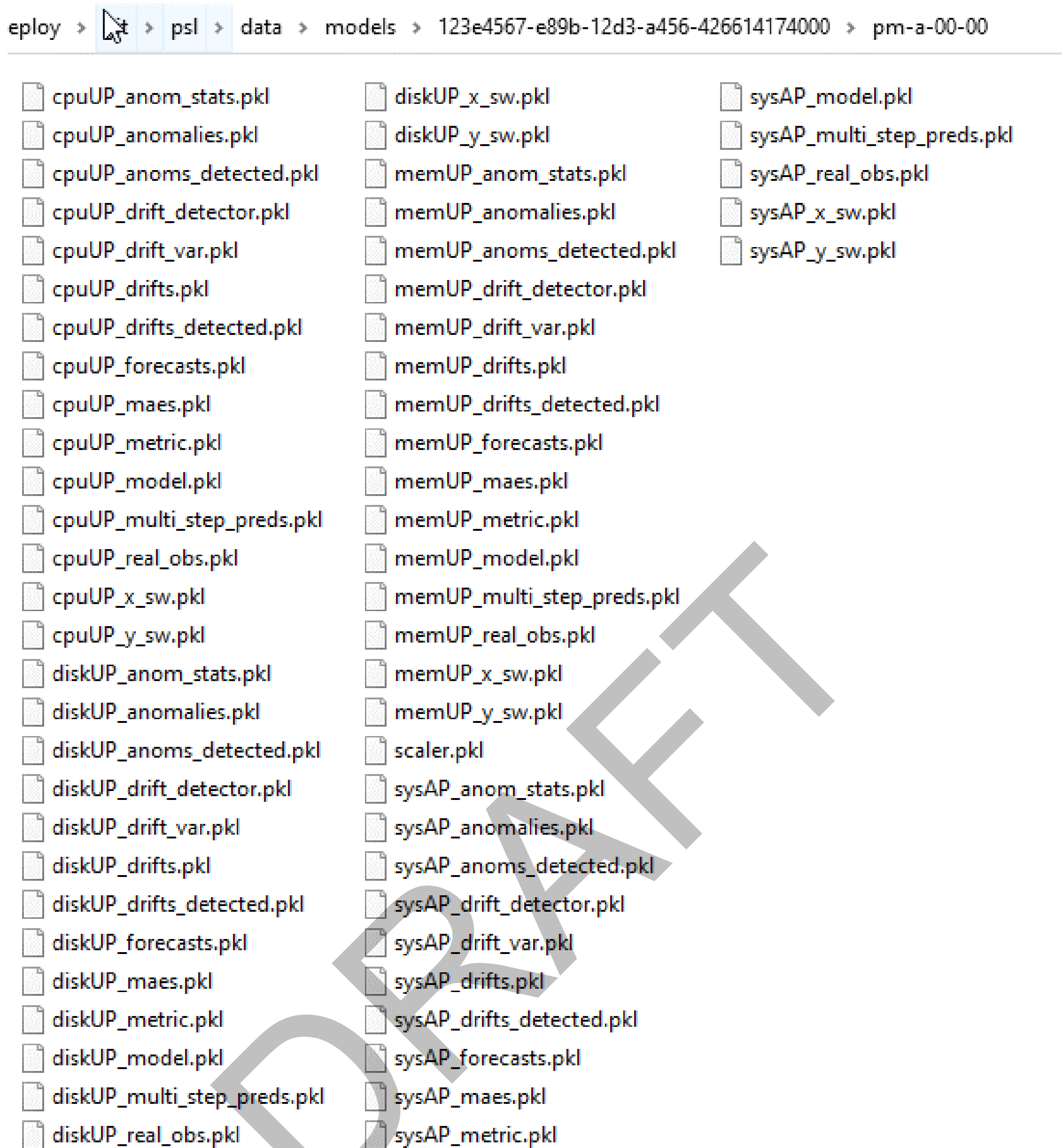


Figure 18 – Self-learning workflow diagram.

After the initial model has been generated, each follow up metric value is used in the online learning and prediction algorithm (6) that also considers anomalies and drifts. Besides, that information is stored back in the time series database (7) for its latter usage in the metrics' visualization (8) and alerts' issuing for self-healing procedures.

DRAFT



DRAFT

Figure 19 – PSL models stored for each DOML element.

The third functionality is the **self-healing**. It reacts to situations detected in the performance monitoring, applying different strategies (Figure 20). The overall flow starts with the reception of events (1) from the monitoring components, specifically performance and security.

Once an event is received (2) through an asynchronous method it is queued for processing. An executor process takes events from the queue and process them. For each event received a strategy to apply is identified (3). The strategies to apply for each event are selected based on three sources with different priority on a deployment basis.

- Higher priority are the strategies specified in the DOML, these are applicable on a deployment basis
- Normal priority are the strategies predefined in the self-healing component
- Default is the notify strategy

(cc) BY-SA

Then each strategy is executed following its own flow, which may involve the communication with external components (4) such as PRC.

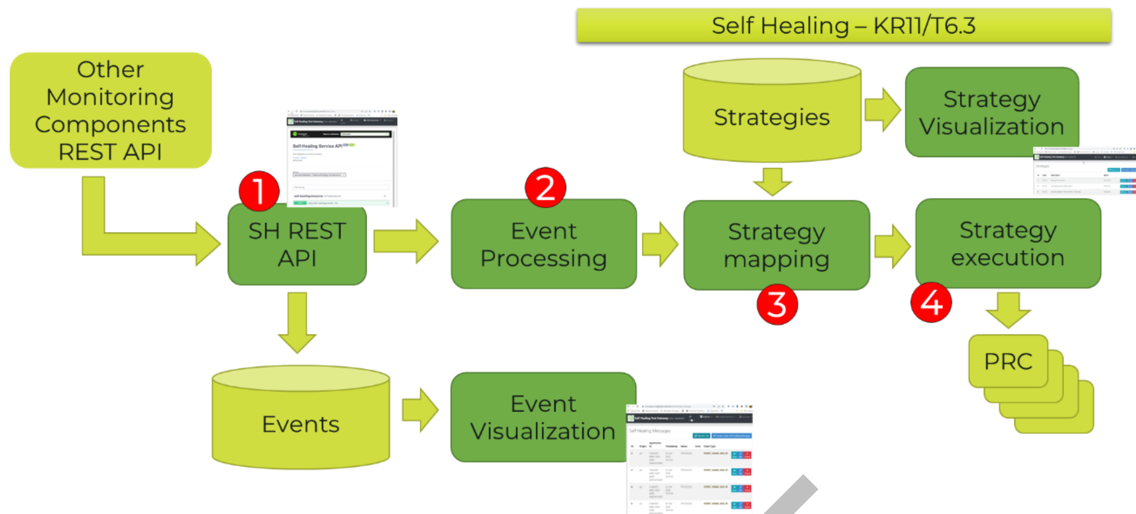


Figure 20 – Self-healing internal workflow.

Based on the strategy, different workflows and interactions will be executed. The different workflows defined are described in the sequence diagram in Figure 21.

- *Notify*: sends a message to the PIACERE administrator, unless redefined based on monitoring rules
- *Redeploy*: it will request the PRC to redeploy the deployment
- *Scale*: it will modify the DOML to scale the infrastructure following the provided configuration.
- *Ansible*: it will execute an Ansible playbook for the deployment. The playbook to be executed.

The following sequence diagram describes how the different strategies work and the interaction required with other components.

DRAFT

(cc) BY-SA

Figure 21 – Self-healing sequence diagram.

In the following lines we will detail the *scale* scenario. As it was already indicated, the default behavior for any event that arrives to the self-healing component is to notify. For example, if we have the DOML infrastructural definition shown in Figure 22, PIACERE will include performance and security agents that will send metrics to the monitoring infrastructure.

DRAFT

```
infrastructure infra {
    vm igw_vm {
     os "Ubuntu-Focal-20.04-Daily-2022-04-19"
     size "medium"

     mem_mb 4096.0

     iface igw_vm_oam {
       belongs_to subnet_oam_igw
     }

     iface igw_vm_net1 {
       belongs_to subnet_net1_igw
     }

     iface igw_vm_net2 {
       belongs_to subnet_net2_igw
     }

     credentials ssh_key

   }
```

Figure 22 – DOML with default monitoring activated.

The monitoring infrastructure will apply the default alerting scheme over those metrics and the default notify strategy will be applied.

For example, if at runtime the memory is over 70% during more than 5 minutes, which is a default alerting scheme for memory (as explained above). Self-healing receives an event from the monitoring. Then, as that deploy has no defined processing for that event, it will apply the default strategy. As described above the default strategy is notify.

But designer may want to apply a more refined and possibly proactive behaviour for that scenario. The way to do that, is through monitoring rules.

DRAFT

(cc) BY-SA

```
infrastructure infra {
    vm igw_vm {
      os "Ubuntu-Focal-20.04-Daily-2022-04-19"
      size "medium"

      mem_mb 4096.0

      iface igw_vm_oam {
         belongs_to subnet_oam_igw
      }

      iface igw_vm_net1 {
         belongs_to subnet_net1_igw
      }

      iface igw_vm_net2 {
         belongs_to subnet_net2_igw
      }

      credentials ssh_key

      monitoring_rules mem_usage_over_70    (2)

   }

      monitoring_rule mem_usage_over_70 {    (1)
      cond "mem_used_percent > 70"
      // A string attribute to specify the name of the monitoring strategy this rule will use.
      // alternatively we can use a json object representing grafana_provisioned_alert_rule
      strat "scale"
      // An optional string attribute to specify parameters valid for the given monitoring strategy.
      config "{
         'ignore_doml_checkings':[ 'iop', 'iac_scan_runner', 'model_checker']
         'memory':
               {
               'upper_limit_mb': '16384.0',
               'intervals_mb': '512.0',
               }
         }"
      }
```

Figure 23 – DOML with explicit monitoring configuration.

The monitoring rules (Figure 23) are specified in the DOML and received by the self-healing component with the activation of the deployment at the top of the sequence diagram in Figure 21. This configures a specific behaviour (1) for that DOML element (2) in the self-healing component.

Now, when the event arrives from the monitoring component the scale strategy will be applied instead of the default strategy. The first thing that will happen is that the DOML will be modified following the strategy configuration. Different configurations are supported such as Memory and CPU. In this case it will modify DOML by adding additional memory.

The strategy configuration in the monitoring rule may define some limits, as in the case of Figure 23 where an upper limit is defined for memory. In case that limit is reached and the strategy cannot be applied the workflow stops and a message is added to the deployment dashboard.

If a new DOML is produced, then a series of checks are performed following the same manual flow that is applied in the manual design time. First, IOP is checked to verify if we are still in the optimized solution. If not, the scale process is stopped.

This behaviour can be modified by adding IOP to the 'ignore_doml_checking' array in the monitoring rule configuration.

The same is done with model checker, and again, if the DOML designer wants, (s)he can ignore the DOML checking step.

DRAFT

(cc) BY-SA

After that ICG is called to get the new IaC code, which is then sent to the IaC scan runner. Again, if the DOML designer wants, (s)he can specify to ignore the IaC scan runner results.

Finally, a new bundle is sent to the PRC for execution, and that finishes the scale strategy workflow.

In the case of the *ansible* strategy, the configuration will indicate the playbook to be executed and its parameters. For the moment the only supported ansible-playbook params are *extra_vars* and *play_book*. The playbook should be provided with the assets of the bundle. The next Figure 24 shows a monitoring rule with the *ansible* strategy, where the specific configuration can be seen.

```
monitoring_rule cpu_usage_over_90 {
    cond "cpu_used_percent > 90"
    // A string attribute to specify the name of the monitoring strategy this rule will use.
    // alternatively we can use a json object representing grafana_provisioned_alert_rule
    strat "ansible"
    // An optional string attribute to specify parameters valid for the given monitoring strategy.
    config "{
            'play_book': 'playbooks/reboot.yaml'
            'extra_vars':
               {
                'when': 'now'
               }
            }
            "
}
```

Figure 24 – ansible monitoring rule.

Results of every step are notified in the deployment dashboard of the self-healing component that can be accessed from the IDE.

The user requirements satisfied by this final version are described in Table 3. All these requirements have been obtained from the *PIACERE WP2 Requirements* internal document.

Table 3 – Monitoring related user requirements from WP2.

Req ID	Description	Status	Requirement Coverage at M30
REQ11	The learning algorithm (anomaly and drift) should be executed as fast as possible as it should provide an outcome before more data arrives.	completed	A REST API has been provided and deployed to be a single point of entry for the configuration of the PIACERE monitoring, self-learning and self-healing components each time that a deployment is requested to the PIACERE runtime controller. The whole Performance Self-Learning component has direct access to the data in the Performance Monitoring database. At the moment execution is fast enough in the application on the use cases.
REQ16	Runtime security monitoring should contribute to mitigation actions taken when considering plans and strategies for runtime self-healing actions	Completed	Basic mitigation strategies have been defined notify, redeploy, scale and ansible in the last period. Besides we have introduced the concept of monitoring rules at DOML level that allow to modify that behaviour at DOML level.

BY-SA

REQ17	Seamless security monitoring deployment *Deployment of runtime security monitoring should happen seamlessly or with minimal effort and configuration required by the user.*	completed	A REST API has been provided and deployed to be a single point of entry for the configuration of the PIACERE monitoring, self-learning and self-healing components each time that a deployment is requested to the PIACERE runtime controller.
REQ46	IOP focused infrastructure metrics *The monitoring component shall gather metrics from the instances of the infrastructural elements at run time. These metrics need to be related to the NFR and accessible to the IOP (through the dynamic part of the infrastructural catalogue).*	completed	The self-healing will feed the infrastructure element catalogue with information about the instances of the services present on it. To do so each time an event is received DOML is analysed to check if a VM flavor is association with the involved DOML element. If present it finds the flavour as service in the service catalogue if available it updates realtime metrics for the service in the service catalogue.
REQ47	Full monitoring stack *The monitoring component shall include the needed elements in the stack to monitor the infrastructural elements.*	Completed	The Performance Monitoring includes all the elements required to monitor infrastructure elements: The agents to gather the information, the database to store the data, the analysis and presentation layer to show the metrics and follow the thresholds, and the component to configure the deployments.
REQ48	Self-learning focused monitoring *The monitoring component shall transform the real time values into the correct format/type/nature for the self learning component.*	Completed	Real time data is stored, and the performance self-learning prototype is actually capable of consuming that information using the provided interface.
REQ50	Monitor performance, availability, and security *The monitoring component shall monitor the metrics associated with the defined measurable NFRs (e.g. performance, availability, and security through the runtime security monitoring).*	Completed	The monitoring components covers the monitoring of performance and availability requirements. It provides default monitoring capabilities as well as customised monitoring capabilities through monitoring rules.
REQ51	Deployment non-functional requirements tracking *The self-learning component shall ensure that the conditions are met (compliance with respect to SLO) and that a failure or a non-compliance of a NFRs is not likely to occur. This implies the compliance of a predefined set of non-functional requirements (e.g. performance).*	Completed	The component will forward all the necessary information to the self-learning components to be able to track the infrastructure related non-functional requirements. We have defined a predefined set of thresholds based on the state of the practice experiences. We have implemented a simplification of the scales of the metrics in order to refactor the

DRAFT

			notifications base on those simplified thresholds. In case different monitoring threshold are required we have implemented the monitoring rules mechanism to allow this refinement.
REQ52	Monitored data based self-learning *Self-healing shall consume the data monitored and store it in a time-series database to create discriminative complex statistical variables and train a predictor, which will learn potential failure patterns in order to prevent the system from falling into an NFR violation situation.*	Completed	The Performance Monitoring currently provides the time series database for the usage by the performance self learning component. This covers a part of this requirement, the other part is covered by the performance self learning component. Dashboards for self-learning have been implemented and are accessible from the IDE for each deployment. Thanks to the integration of the Performance Monitoring database access implementation, the Performance Self-Learning component is able to consume the data in an incremental way and to create the necessary variables.
REQ72	monitoring user interface *The runtime monitoring component should provide an UI for the end users to see the monitored resources and the corresponding metrics/NFRs in real time.*	Completed	The current version of the Performance Monitoring includes a graphical user interface that renders the information coming from the time series database. We have introduced a deployment-based dashboard that includes information related to the NFR thresholds coming from the DOML specification.
REQ93	Self-healing should classify the events notified	Completed	We receive notifications from some of the monitoring elements based on the identification of the monitoring event and applying a classification scheme a strategy is assigned and executed. If the identifier of the monitoring event has no strategy associated notify strategy is applied. The association identifiers to strategies comes from two sources: pre-configuration of default strategies and deployment specific rules based on monitoring rules.
REQ94	Self healing component shall inform the run time controller about the different components to orchestrate (the workflow to be executed)	Completed	Self-healing request the redeployment of deployments to the runtime controller when appropriate. More concretely with redeploy, scale and ansible strategies.
REQ97	The Self-Healing components provide feedback on the DOML code, without doing automatic writes. The end user can choose to accept or not	Completed	Self-healing has notification procedures in case the DOML modifications do not match some requirements.

DRAFT

CC BY-SA

	the feedback received. (ex REQ56&75)		For scale scenarios IOP is queried with the change and if the suggestion changes a notification is provided. Besides, the scale scenarios require configuration to establish limits on how much the resources can grow.

The internal requirements satisfied by this latest version are described in the Table 4. All these requirements are as well polished and adapted as the project advances.

Table 4 – Performance Monitoring related internal requirements.

Description	Status	Requirement Coverage at M30
Add code into the project source repository	Completed	The repository has been created and the code is being uploaded regularly • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pm/-/blob/y3 • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/mc/-/blob/y3 • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pmc/-/blob/y3 • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pma/-/blob/y3 • https://git.code.tecnalia.com/piacere/public/the-platform/self-learning/-/blob/y3 • https://git.code.tecnalia.com/piacere/public/the-platform/self-healing/-/blob/y3 https://git.code.tecnalia.com/piacere/public/agents/pma-playbook/-/tree/y3
Implement REST API specification	Completed	The final version of the OpenAPI has been defined and put under configuration control • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pm/-/blob/y3/git/pmc/openapi.yaml • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/mc/-/tree/main/src/mc/openapi.yaml • https://git.code.tecnalia.com/piacere/public/the-platform/self-learning/-/blob/y3/src/psl/openapi/openapi.yaml • https://git.code.tecnalia.com/piacere/public/the-platform/self-healing/-/blob/y3/openapi.json The OpenAPI have been implemented and put in integration.
Implement specification first approach	Completed	In order to speed-up the implementation of changes derived from the expected evolution of the REST API, we have implemented a specification first approach with OpenAPI generator. Besides, the usage of OpenAPI generator brings additional benefits in the sense of introduction of good practices in structuring and configuring the code.
Prepare for deployment	Completed	In order to ensure that we are prepared to deploy the component in the integration environment we have integrated this component with the remaining monitoring components in a Docker-compose file that includes a reverse proxy to receive all the requests using secure standard HTTPS protocol.

DRAFT

(cc) BY-SA

		https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pm
Provide fast deployment alternative for deployment, testing and evaluation	Completed	To allow a seamless infrastructure requirements free alternative to test this component we have provided a docker compose based build and deployment option. This docker compose based option has been used as the basis for the development during the life of the project. Besides, we have tested it in a Vagrant based deployment option. This reduces the list of software requirements to two: VirtualBox and Vagrant. These two tools (VirtualBox and Vagrant) are available for most of the operating systems: Windows, Mac, Linux, BSD, …
Include usage documentation	Completed	We have included usage documentation at different levels: • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pm/-/blob/y3 • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/mc/-/blob/y3 • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pmc/-/blob/y3 • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pma/-/blob/y3 • https://git.code.tecnalia.com/piacere/public/the-platform/self-learning/-/blob/y3 • https://git.code.tecnalia.com/piacere/public/the-platform/self-healing/-/blob/y3 • https://git.code.tecnalia.com/piacere/public/agents/pma-playbook/-/tree/y3 We update continuously the documentation as we advance in the coding and pre-integration of the monitoring components
Unitary test	Planned	We will include unitary tests as part of the exploitation strategy.
Integration test	Completed	We have completed the end-to-end deployment scenario for both the demo project and for the use cases.
Continuous integration	Completed	Continuous integration has been implemented based on gitlab-ci and integrated with the rest of components of the PIACERE framework • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pm/-/blob/y3/.gitlab-ci.yml • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/mc/-/blob/y3/.gitlab-ci.yml • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pmc/-/blob/y3/.gitlab-ci.yml • https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pma/-/blob/y3/.gitlab-ci.yml • https://git.code.tecnalia.com/piacere/public/the-platform/self-learning/-/blob/y3/.gitlab-ci.yml • https://git.code.tecnalia.com/piacere/public/the-platform/self-healing/-/blob/y3/.gitlab-ci.yml There are some components that do not require integration as they are integrated in other components. Specifically, the monitoring agents. • https://git.code.tecnalia.com/piacere/public/agents/pma-playbook/-/tree/y3?ref_type=heads

DRAFT

(cc) BY-SA

		Which are integrated in the ICG using submodule mechanisms. This code is packaged as part of the ICG continuous integration.

2.3 Main Innovations

Main innovations introduced during this period are:

- Establish a default simplified and easy to understand monitoring scheme based on four metrics, all of them using percentual scale. Where in general the greater the percentage is we worst the situation is.
- Definition of the infrastructure availability metric based on the gaps in the reception of metrics from the monitoring agents deployed in the infrastructure elements that build up the setup deployments.
- DOML monitoring rules have been defined and implemented to allow the adaptation and extensibility of the monitoring features to the project needs. On the one hand, with these rules we can customize the conditions under which events are sent to the self-healing. On the other hand, providing the possibility to modify the behaviour of some of the healing strategies.
- Ansible strategies have been introduced for the execution of ansible playbooks on the elements of the infrastructure, allowing a high customization of the framework.
- Catalogue feedback have been implemented to add information about the events captured in the monitoring framework, for the individual monitored elements.

DRAFT

CC BY-SA

3 KR12 Runtime security monitoring overview

KR12 is related with tasks T6.2, T6.4.

KR12 – Runtime security monitoring provides verification and detection of security violations at runtime.

KR12 provides a monitoring system capable of detecting security-related events and incidents in the deployed application's environment. It is (to the extent possible) deployable automatically and notifies users about security alerts.

3.1 Changes in v3

In the reported period the work has been done mostly on implementation updates of the Security Monitoring Deployment and Security Monitoring Controller. Moreover, there were updates to the Security Monitoring Agents.

Changes in the Security Monitoring Deployment:

- Integration of custom-decoders and rules, also supporting security self-learning monitoring component
- Supporting IEM with the deployment of Security Monitoring infrastructure.

Changes of the Security Monitoring Controller:

- Updated Event reports helper – support for the self-learning mechanism

Changes of the Security Monitoring Agents:

- Support deployment to different architectures (CentOS, debian, Docker environment)

For the components listed above components e have supported infrastructure tasks (updates to the CI/CD).

3.1.1 Integration of custom-decoders and rules for self-learning

Wazuh allows you to create your own custom rules and decoders. This enables you to introduce additional detection logic that is tailored to specific requirements. You can define specific conditions, keywords, or patterns to identify security events or anomalies that are relevant to your environment. By creating custom rules and decoders, you can expand the coverage and depth of detection provided by Wazuh. The idea of using custom-decoders and rules for the self-learning purposes is that the security infrastructure (Wazuh) defines custom rules and decoders in a way, so that all raw logs from the agent are aggregated towards the servers and collected within the dedicated Elasticsearch. This is not being done by default by Wazuh.

Local Decoder residing in configuration of Wazuh's instance is located in the file: Config/local_decoder.xml:

```
<decoder name="allow_all">
<prematch>\.</prematch>
</decoder>
<decoder name="allow_all">
<parent>allow_all</parent>
<regex type="pcre2">(?i)(.*)(EventChannel)(.*)</regex>
<order>data,action,extra_data</order>
</decoder>
```

Local Rule residing in the configuration of Wazuh's instance - Config/local_rules.xml:

DRAFT

BY-SA

```
<rule id="101010" level="9">
<regex type="pcre2">.*</regex>
<description>Allow all logs not caught by inbuilt rules</description>
</rule>
```

These rules enable infrastructure to gather all raw logs from infrastructure to be injected into the monitoring services and index that is later used by self-learning mechanisms. The raw logs are then used by the LOMOS (LOg MOnitoring System, see section 3.2.1) system in order to do the model for the anomaly detection and process the logs on top of the raw logs.

3.1.2 Supporting IEM with the deployment of Security Monitoring infrastructure

Modifications related to the deployment of the Security Monitoring infrastructure we provided so that the deployment was made more efficient (and cloud native) supporting different operating system flavours (Centos, Debian, Docker-specifics). Specifically, updates have been made to the sma-playbook (deployment for the agents).

3.1.3 Extensions to DOML to support Security Monitoring and Security Self-Learning

Referring to the section 2.1.4, extensions to DOML have been made from the security monitoring perspective. We have also defined specific DOML rules with respect to security monitoring and security self-learning.

Examples of three security monitoring rules are given below. The first is an example of a monitoring rule notifying user that some anomaly related to security has been detected. "Notify" strategy has been enforced here implying that the user will get a notification about the anomaly being detected.

```
Monitoring_rule sec_anomaly_notify {
      /*
       * A formal string attribute, whose value is dependent on the strategy attribute,
       * that defines the condition that will trigger the monitoring.
       */
      cond "sec_anomaly > 0.5"
      // A string attribute to specify the name of the monitoring strategy this rule 
will use.
      Strat "notify"
      // An optional string attribute to specify parameters valid for the given 
monitoring strategy.
      Config "Security anomaly detected. Please, check the logs of security monitoring."
}
```

The following is another example of a rule about notification to the user that Security Component Analysis' threshold has been reached (or smaller to the one being acceptable):

```
      monitoring_rule sec_sca_scan_rule {
            cond "sca.policy < 0.5"
            strat "notify"
            config "SCA summary: sca.policy : Score less than 50% (sca.score). SCA scan 
threshold is below the accepted level. Please consider hardening the OS."
      }
```

The last is an example of an ansible strategy that enforces running an Ansible script towards the target infrastructure:

```
      monitoring_rule sec_port_rule {
            cond "sec_unattended_port"
            strat "ansible"
            config " { 'playbook': 'playbook/harden_ports_fw.yml', 
            'extra_vars': { 'allow_ports': [ 80, 443 ] }}"
      }
```

DRAFT

BY-SA

In the last example we assume that the "harden_ports_fw.yml" Ansible playbook is made available (in the assets folder) and is specific to this target infrastructure.

3.1.4 Updated Event reports helper

Security Monitoring Controller has been updated with the capability to upload specific notifications towards Self-Healing endpoint taking into account a specific timeout and threshold which is part of the configuration and the rule's input.

With this in mind, additional configuration attributes have been introduced within the Security Monitoring Controller:

```
SM_POLL_WEBHOOK_URL = https://sh.ci.piacere.digital.tecnalia.dev/api/self-healing/notify
SM_POLL_WEBHOOK_USERNAME = admin
SM_POLL_WEBHOOK_PASSWORD = xxxxx
SM_POLL_TIMEOUT = 10
SM_POLL_THRESHOLD = 9
```

This configuration sets the service's API towards the reports are pushed. Timeout and threshold effect the mechanisms w.r.t. the default settings (in case no specific thresholds are set, these settings are taken into account). Basic authentication is supported (as supported by Security Self Healing).

Event reports helper within Security Monitoring Controller embeds a process with a time-window of the size of SM_POLL_TIMEOUT seconds). As soon as the security monitoring rules are being triggered within this time window, the events are being sent to the SM_POLL_WEBHOOK_URL using the scheme from Wazuh events.

Log entry related to a security monitoring system - explanation of the key fields from the report:

- Agent: Represents the agent or device that generated the log entry. It includes information such as the agent's IP address, name, ID, and labels.
- Rule: Contains information about the rule that was triggered. It includes details like the number of times the rule has fired, the rule's level, and various compliance standard details tagged associated with the rule.
- Full Log: Presents the complete log message or output generated by the agent or device.
- Manager: Represents the name of the security management system or manager.
- Decoder: Indicates the name of the decoder responsible for parsing and interpreting the log message.
- Input: Specifies the type of input received, in this case, it is a log.
- @timestamp: Indicates the timestamp when the log entry was created.
- Location: Describes the location or context of the log entry, in this case, it refers to "netstat listening ports".
- _id: Represents the unique ID assigned to the log entry.

In this specific log entry, the rule with ID "533" from Wazuh was triggered because the status of the listened ports (as determined by the "netstat" command on the infrastructure) changed. It indicates that a new port was opened or closed. The log message shows the difference between the current and previous states of the listened ports.

Example of an event reported (detection of an unattended port being opened on the infrastructure):


```
{
      "agent": {
            "ip": "172.20.0.29",
            "name": "aa7a325d72fb",
            "id": "001",
```


DRAFT

```
            "labels": {
                  "piacere-deployment-id": "123e4567-e89b-12d3-a456-426614174001"
            }
      },
      "rule": {
            "firedtimes": 1,
            "mail": false,
            "level": 7,
            "pci_dss": [
                  "10.2.7",
                  "10.6.1"
            ],
            "hipaa": [
                  "164.312.b"
            ],
            "tsc": [
                  "CC6.8",
                  "CC7.2",
                  "CC7.3"
            ],
            "description": "Listened ports status (netstat) changed (new port opened or
closed).",
            "groups": [
                  "ossec"
            ],
            "id": "533",
            "nist_800_53": [
                  "AU.14",
                  "AU.6"
            ],
            "gpg13": [
                  "10.1"
            ],
            "gdpr": [
                  "IV_35.7.d"
            ]
      },
      "full_log": "ossec: output: 'netstat listening ports':\ntcp        0      0
127.0.0.11:44783        0.0.0.0:*               LISTEN      -                  \nudp
0      0 127.0.0.11:54828        0.0.0.0:*                           -
",
      "previous_log": "ossec: output: 'netstat listening ports':\ntcp        0      0
127.0.0.11:40069        0.0.0.0:*               LISTEN      -                  \nudp
0      0 127.0.0.11:42571        0.0.0.0:*                           -
",
      "id": "1680702615.19456",
      "timestamp": "2023-04-05T13:50:15.832+0000",
      "previous_output": "Previous output:\nossec: output: 'netstat listening
ports':\ntcp        0      0 127.0.0.11:40069        0.0.0.0:*               LISTEN
-                  \nudp        0      0 127.0.0.11:42571        0.0.0.0:*
-                  ",
      "manager": {
            "name": "wazuh-manager"
      },
      "decoder": {
            "name": "ossec"
      },
      "input": {
            "type": "log"
      },
      "@timestamp": "2023-04-05T13:50:15.832Z",
      "location": "netstat listening ports",
      "_id": "bU6uUYcBIjeGvwXZtzW_"

}
```


3.2 Functional description and requirements coverage

KR12 provides functionalities focused to support monitoring and self-healing capabilities with respect to security events/metrics detected on the infrastructure. Similarly to KR11, the objective is to implement similar main functionalities:

- Basic security monitoring

BY-SA

- Self-learning
- support Self-healing process

3.2.1 Functional description

Since **security monitoring** is very different in PIACERE comparing to runtime monitoring, mostly due to the nature of events being detected and analysed (the events are based on events detected in the logs, and are not numerical metrics), in order to achieve the result, the monitoring facility is based on log-based monitoring infrastructure. Basic security monitoring also provides specific rules that are taken into account while triggering the self-healing process.

The monitoring agents can detect the following type of events:

1. **Listened Ports**: The agent monitors changes in the status of listened ports. It can detect if new ports are opened or closed.
2. **Rootkit Files and Trojans**: The agent performs rootcheck scans to detect rootkit files and trojans on the system.
3. **System Inventory**: The agent uses the Syscollector module to collect system inventory information such as hardware details, operating system information, network configurations, installed packages, and running processes.
4. **File Integrity**: The agent performs file integrity monitoring using the syscheck module. It monitors specified directories and files for any changes or modifications, ensuring the integrity of critical system files.
5. **Log Analysis**: The agent performs log analysis on specific files and commands. The provided XML configuration includes log analysis for the output of the `df -P` command, `netstat` command for listening ports, and the `last` command for system login information.

These metrics allow the Wazuh Agent to monitor and detect changes, anomalies, and potential security threats in the system's network, file system, system configuration, and log files. By analyzing these metrics, the agent can provide valuable insights and trigger alerts or automated response actions when necessary.

DRAFT

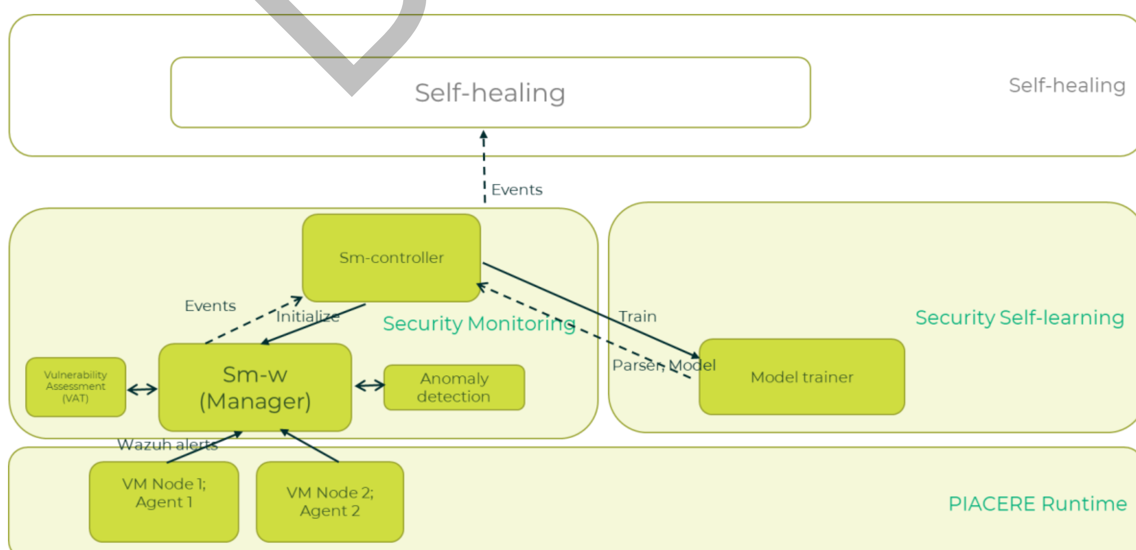


Figure 25 – High-level architecture diagram of Security Monitoring components.

(cc) BY-SA

The Security monitoring system consists of subsystems (Wazuh deployment – manager and agents – with specific components for data transformation) collecting data (Figure 25, left hand-side) in order to provide values for security metrics. The system stores (1) data aggregated by the (security) monitoring system and (2) data generated by underlying anomaly detection system using dedicated ELK stack. The ElasticSearch storage of the is located in the SM-W (The Manager in the picture above).

Self-learning is based on the system named LOMOS[2] (Log Monitoring System, Figure 25, right hand-side name Security Self-learning) – it is an ML-based anomaly detection solution. Self-learning component is hooked on the SM-W component from the figure above in order to track raw logs from the storage system. It compares the logs towards the model which is pre-built (in the learning phase of the use). Security Self-learning's role in PIACERE is to provide a second layer of analysis of gathering data and metrics. The Self-healing part is outsourced to the existing self-healing PIACERE component that is capable of triggering specific strategies based on the configured basic security monitoring rules and Self-learning security monitoring rules.

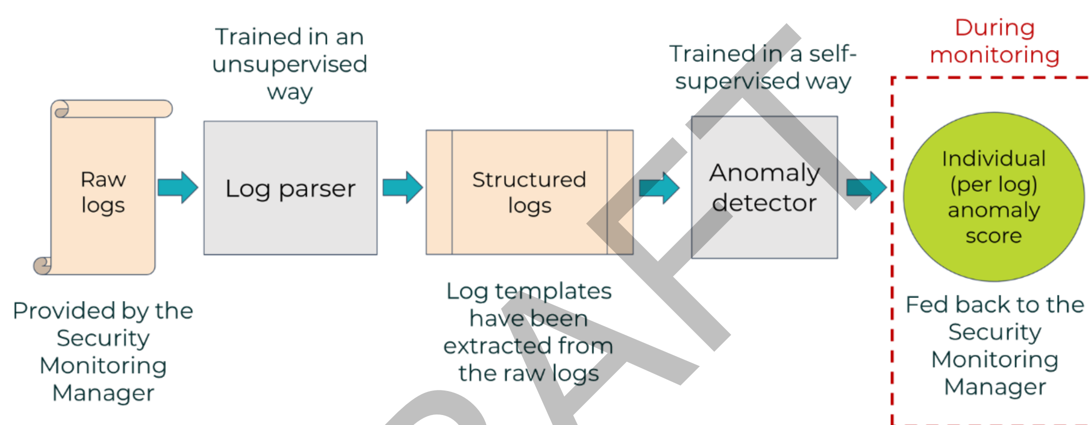


Figure 26 – Security Self-learning approach based on LOMOS- anomaly detection workflow from logs.

Figure 26 presents the approach used by LOMOS in order to detect anomalies from the logs. LOMOS operates on raw logs without pre-processing and focuses on two main objectives: learning patterns and identifying anomalous behaviour. To achieve this, LOMOS employs an algorithm that seeks to establish structure within the logs by identifying log templates that can be matched against the log data.

The underlying algorithm of LOMOS (based on the extension of LogBERT[3] algorithms, using Drain[4] for log template parsing), is designed to extract key parameters such as IDs, services, ports, and other relevant information. These parameters play a critical role in transforming the initially unstructured logs into structured log templates. The log templates are organized hierarchically, following a tree-like structure. Within this structure, certain elements remain constant across multiple logs, while other elements are variable and represented by placeholders or wildcards. These wildcards allow for flexibility in capturing and analysing logs with varying parameters from log to log.

[2] J. Antić et al., "Runtime security monitoring by an interplay between rule matching and deep learning-based anomaly detection on logs," 2023 19th International Conference on the Design of Reliable Communication Networks (DRCN), Vilanova i la Geltru, Spain, 2023, pp. 1-5, doi: 10.1109/DRCN57075.2023.10108105.
[3] H. Guo, S. Yuan, and X. Wu. Logbert: Log anomaly detection via bert, 2021
[4] P. He, J. Zhu, Z. Zheng, and M. R. Lyu. Drain: An online log parsing approach with fixed depth tree. In 2017 IEEE international conference on web services (ICWS), pages 33–40. IEEE, 2017.

BY-SA
DRAFT

3.2.2 Requirements coverage

Table 5 – Security Monitoring requirements from WP2.

Req ID	Description	Status	Requirement Coverage at M30
REQ14	Runtime security monitoring must provide monitoring data from the infrastructure's hosts w.r.t. security metrics	completed	Security Monitoring Controller provides API call in order to get the alerts/events from the stored database.
REQ15	Runtime security monitoring could provide monitoring data from the application layer (infrastructure's guest) w.r.t. security metrics	completed	This is possible through the configuration of the Security Monitoring Manager (specifically, Wazuh configuration).
REQ16	Runtime security monitoring should contribute to mitigation actions taken when considering plans and strategies for runtime self-healing actions	completed	The integration with the self-healing components – through DOML you are capable of expressing different self-healing actions
REQ17	Deployment of runtime security monitoring should happen seamlessly or with minimal effort and configuration required by the user.	completed	The deployment is integrated with the complete deployment of a specific DOML.
REQ18	Runtime security monitoring must be able to detect different types of metrics in run-time: integrity of IaC configuration, potential attacks to the infrastructure, IaC security issues (known CVEs of the environment).	completed	The data of these metrics are already available in the Security Monitoring infrastructure. However, this is possible through the configuration of the Security Monitoring Manager (specifically, Wazuh configuration). The configuration needs to be provided through the configuration step.
REQ19	Runtime security monitoring and alarm system (self-learning) integration must be implemented.	Completed	The integration between the monitoring in self-learning (LOMOS) is available.
REQ21	Runtime security monitoring and Runtime monitoring infrastructure should be integrated with minimal extensions.	completed	The integration is done through the deployment of the Security Monitoring Agents and their deployment code.
REQ50	The monitoring component shall monitor the metrics associated with the defined measurable NFRs (e.g. performance, availability, and security through the runtime security monitoring)	completed	Integration between "expressing NFRs" and configuration of the security monitoring infrastructure through DOML is available.
REQ51	The self-learning component shall ensure that the conditions are met (compliance with respect to SLO) and that a failure or a non-compliance of a NFRs is not likely to occur. This implies the compliance of a predefined set of non-functional requirements (e.g. performance)	completed	The self-learning component of security monitoring can be used to build a specific model to be used for detecting metrics with respect to anomaly detection (anomalies detected on the infrastructure). It is possible to express these metrics and related NFRs through the integration with DOML components and the rest of PIACERE flow (NFRs to be consumed by Security Monitoring).

DRAFT

Table 6 – Security Monitoring related internal requirements.

CC BY-SA

Description	Status	Requirement Coverage at M30
Add code into the project source repository	Completed	The repository has been created and the code is being uploaded regularly https://git.code.tecnalia.com/piacere/private/t64-runtime-security-monitoring/security-monitoring-controller and https://git.code.tecnalia.com/piacere/private/t64-runtime-security-monitoring/security-monitoring-deployment Agents deployment https://git.code.tecnalia.com/piacere/public/agents/sma-playbook
Implement REST API specification	Completed	OpenAPI of security monitoring controlled is made available: https://git.code.tecnalia.com/piacere/private/t64-runtime-security-monitoring/security-monitoring-controller
Implement specification first approach	Completed	In order to speed-up the implementation of changes derived from the expected evolution of the REST API, we have implemented a specification first approach with OpenAPI generator.
Prepare for deployment	Completed	Part of the code provided on the Gitlab. https://git.code.tecnalia.com/piacere/private/t64-runtime-security-monitoring/security-monitoring-deployment https://git.code.tecnalia.com/piacere/public/agents/sma-playbook
Provide fast deployment alternative for deployment, testing and evaluation	Completed	Part of the code provided on the Gitlab.
Include usage documentation	Completed	Part of the code provided on the Gitlab.
Unitary test	Planned	We will include unitary tests as part of the exploitation strategy.
Integration test	Completed	We have completed the end-to-end deployment scenario for both the demo project and for the use cases.
Continuous integration	Completed	Continuous integration has been implemented based on gitlab-ci and integrated with the rest of components of the PIACERE framework.

DRAFT

3.3 Main Innovations

The main innovations of the KR12 are:

- Seamless deployment of security monitoring infrastructure together with infrastructure expressed through DOML
- Integration of the self-learning (log-based) anomaly detection system with security monitoring infrastructure
- Integration of anomaly detection system with self-healing systems (triggering specific self-healing strategies based on detected anomalies)
- Custom decoders and rules supporting security self-learning components with raw log events for building the model and for the inspection process

CC BY-SA

4 Overview of preliminary experiments

During this period, we have advanced in the deployment and un-deployment of IaC generated by the ICG containing the monitoring agents. The overall procedure with all the use cases has been:

- Identification of input DOML
- Execution of the DOML with ICG to generate the IaC
- Ensure the provider details
- Execute the IaC using the PRC
- Achieve the successful deployment of the IaC
- Feedback to the used components

The experiments carried out for the monitoring component included:

- Demo nginx
- Posidonia
- Agents integration at IEM
- Agents with ICG
- SIMPA
- Ericsson

Demo nginx was used initially to check the deployment of the performance monitoring agents over the existing deployment that was including Terraform and nginx IEM stages. We added the PIACERE monitoring stages Figure 27.

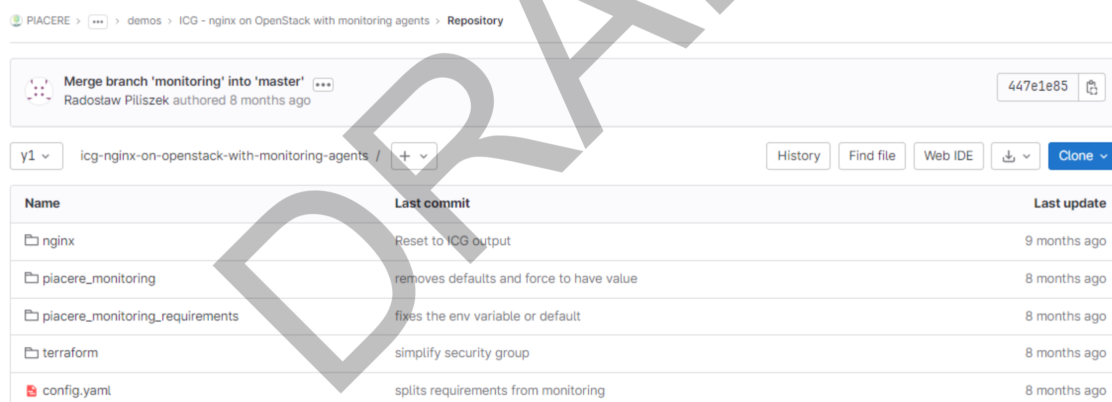


Figure 27 – nginx demo first monitoring integration.

From that experiment we learnt some lessons:

- The usage of IEM with the creation of machines required some time
- The installation of the performance monitoring agents was subject to be merged in one single stage, reducing the maintenance complexity
- There were issues with the clean-up of the resources, as the key pair were not removed when manually issuing the *terraform destroy* command.

Posidonia was the next experiment as shown in Figure 28.

DRAFT

CC BY-SA

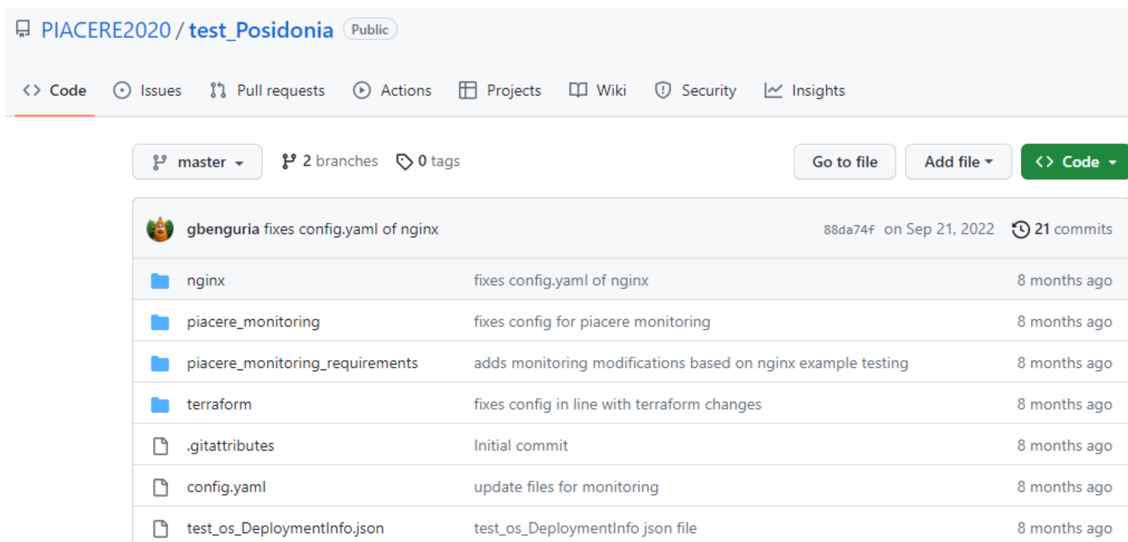


Figure 28 – Posidonia demo monitoring integration.

The experiment was used with two aims: to understand the deployment process with the use case and its interaction with the IDE, and to improve the monitoring agents. From the experiment we implemented:

- The monitoring controller link to customize the monitoring components, that was called by the PRC
- The IDE link to the monitoring dashboard
- Performance self-learning specific dashboard

From the experiment we also improved the monitoring agents, reducing them to a single stage.

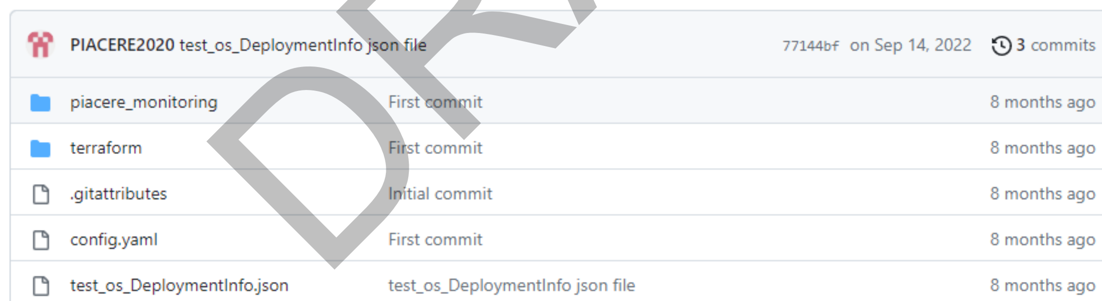


Figure 29 – Posidonia performance monitoring agent improvement.

Next experiment, agents integration at IEM Figure 30, was centred in the integration of both performance and security agents. For this experiment and due to the required agility in this endeavour we switched from a Terraform based approach to a Docker based approach in which we used Docker containers as virtual machines. To do so we developed Docker images with SSH services running in a privileged mode to allow full control with *systemctl*, and be as much similar as possible to virtual machines.

CC BY-SA

```
# Demo agents at iem with dockers

The objective is to be able to check the agents at iem with dockers. Without requiring to create virtual machines that takes some time to be created. Besides, it requires some after demo cleaning activities.

It requires to have the iem working in the same server in the traefik network in order to be able to access the demo server that is simulated by a container in that network.

NOTE: This is a submodule based repo remember issuing "git submodule update --init --recursive" to get the submodules.

``` bash
docker login -u <user> -p <password> optima-piacere-docker-dev.artifact.tecnalia.com
docker network create traefik_network
export ADMIN_PASSWORD=piacerePassword && \
docker run --name iem --rm --network traefik_network --env API_KEY=$ADMIN_PASSWORD --env INFLUXDB_ADDR=https://influxdb.pm.ci.piacere.digital.tecnalia.dev --env INFLUXDB_BUCKET=bucket --env INFLUXDB_ORG=piacere --env INFLUXDB_TOKEN=piacerePassword --env LOG_LEVEL=INFO --env WAZUH_MANAGER_HOST=wazuh-manager.sm.ci.piacere.digital.tecnalia.dev --env WAZUH_MANAGER_PORT=1514 optima-piacere-docker-dev.artifact.tecnalia.com/wp5/iem:y2
```

In another terminal, we can run the demo in the same network.

``` bash
export ADMIN_PASSWORD=piacerePassword && \
docker-compose kill && \
docker-compose down --volumes --remove-orphans && \
docker-compose build && \
docker-compose  up -d --no-build --remove-orphans && \
docker-compose logs --tail=100 -f
```

Figure 30 – Agents at IEM.

DRAFT

From this experiment we obtain a fast and easily replicable way to check monitoring agents from the IEM. It was fast because containers are created much faster than virtual machines. It was easily replicable because only one Docker enabled machine is required to replicate the experiment.

Besides, from this experiment we understood how to configure the IEM in the integration environment, and we identified some improvements in the IEM logging capabilities that were implemented and highly improved the understanding of the errors that IEM finds when applying Ansible playbooks.

Next experiment, agents with ICG Figure 31, was centred in going one step forward and generate the IaC using the DOML and apply it through the IEM.

(cc) BY-SA
```

```
# Demo agents at icg with dockers

The objective is to be able to check the agents at icg with dockers. Without requiring to create virtual machines that takes some time to be created. Besides, it requires some after demo cleaning activities.

It requires to have the icg working in the same server in the traefik network in order to be able to access the demo server that is simulated by a container in that network.

NOTE: This is a submodule based repo remember issuing "git submodule update --init --recursive" to get the submodules.

``` bash
docker login -u <user> -p <password> optima-piacere-docker-dev.artifact.tecnalia.com
docker network create traefik_network
export ADMIN_PASSWORD=piacerePassword && \
docker run --name iem --rm --network traefik_network --env API_KEY=$ADMIN_PASSWORD --env INFLUXDB_ADDR=https://influxdb.pm.ci.piacere.digital.tecnalia.dev --env INFLUXDB_BUCKET=bucket --env INFLUXDB_ORG=piacere --env INFLUXDB_TOKEN=piacerePassword --env LOG_LEVEL=INFO --env WAZUH_MANAGER_HOST=wazuh-manager.sm.ci.piacere.digital.tecnalia.dev --env WAZUH_MANAGER_PORT=1514 optima-piacere-docker-dev.artifact.tecnalia.com/wp5/iem:y2
```
docker run --name icg --rm --network traefik_network optima-piacere-docker-dev.artifact.tecnalia.com/wp3/icg:y2
```

In another terminal, we can run the demo in the same network.

``` bash
```

Figure 31 – Agents through ICG.

DRAFT

From this experiment we identified critical problems in the configuration management of the agents. ICG versions and the latest versions that we were applying were different and the synchronization procedures were not agile enough.

To solve this situation, on the one hand we refactored the ICG to use git submodules Figure 32, on the other hand we changed the time in which the agents are included in the deployment bundle.

PIACERE > ... > T34 ICG-Infrastucture code generator > ICG Controller

updated vSphere Terraform templates,updated files API to output all contents
laurentiu.niculut@hpe.com authored 14 hours ago — 5f1d870b

y3 — t34-icg-controller / templates / ansible / cross-platform /

Name	Last commit	Last update
..		
performance_monitoring @ 88b10d9f	updated vSphere Terraform templates,updated files API to output all con...	14 hours ago
security_monitoring @ 8b06981d	refactors monitoring agents	2 months ago

Figure 32 – Submodule link to monitoring agents.

The submodule approach allows to easily transfer the improvements introduced in the monitoring agents in the ICG by simply doing a git checkout inside the submodule. A side effect of using the same code for deployable IaC and for templates was that some configured by ICG were overwritten when copying the content of the agent submodule. We were including some critical files, i.e. inventory.j2. and ssh_key.j2 in the agents for local testing. We find out that due

CC BY-SA

to the time when ICG was copying the template the generated jinja files were overwritten. Therefore, we changed the order to copy first the template and then generating the files.

Next experiment was centred in the SIMPA use case. For this use case we were asked to deploy the agents in an on-premise cloud, specifically vSphere.

For the implementation of this experiment, we followed a hybrid approach similar to the previous agents with ICG experiment: a Docker based approach for testing the deployment in docker simulated infrastructure and a Terraform based approach for real deployment experience. In this experiment we faced several challenges:

- The hybrid approach
- The easy replication by the use case owners in a separate on-premise cloud infrastructure
- The deployment time, as we planned to show SIMPA how to replicate during a series of short telcos, having as much time as possible for discussions
- The clean-up of resources, as we were using an alternative on premise infrastructure for our deployment

There were some main steps in this scenario:

- Implementation of Docker based scenario
- Modification of their Terraform code
- Introduction of PRC
- Demo to SIMPA
- Adaptation to SIMPA

The first step was to implement a Docker based scenario, the SIMPA scenario was using two infrastructure resources while all our previous experimentations where deploying into one. To clear out all the possible contingencies, starting from our previous experience with agents at IEM, we created a new experiment project SIMPA at IEM Figure 33.

DRAFT

(cc) BY-SA

```
# Simpa at iem with dockers

The objective is to be able to check the simpa at iem with terraform on vcenter.

It requires to have the iem working in the same server in the traefik network in order to be able to access the simpa server that is simulated by a container in that network.

NOTE: This is a submodule based repo remember issuing "git submodule update --init --recursive" to get the submodules.

The repo contains two main docker-compose deployments:
* .env that deploys the simpa in terraform it requires some secrets to be in place to be able to deploy in tecnalia's vcenter
* .env.docker that deploys the simpa in docker it does not requires secrets but it does not deploy in vcenter it uses docker images to simulate the vms. Its propouse is to support the fine tuning of the simpa ansible configuration.

** for the vsphere focussed deployment

NOTE: for this to work it is necessary to setup first the VSPHERE connection variables
export VSPHERE_PASSWORD=*****
export VSPHERE_ALLOW_UNVERIFIED_SSL=*****
export VSPHERE_USER=*****
export VSPHERE_SERVER=*****

``` bash
cd simpa-at-iem-deploy
docker login -u <user> -p <password> optima-piacere-docker-dev.artifact.tecnalia.com
export ADMIN_PASSWORD=piacerePassword
docker-compose pull && \
```

Figure 33 – SIMPA at IEM.

DRAFT

This was interesting to understand what we require from the previous Terraform stage. Once we verified that the agents were deployed in the two instances and reporting measurements to the monitoring dashboard, we shifted to the next step.

The next step is the implementation of the Terraform based deployment. To do this adjustment we used an alternate on-premise infrastructure different from the one that will be potentially used by SIMPA. Both virtualization platforms were technologically similar, as both were vSphere clusters. In this way the Terraform to be applied was similar. Different adjustments were necessary in this stage:

- IEM was modified to accept vSphere credentials
- The Terraform code provided by the use case, was adjusted due to three main differences:
    - The type of infrastructure was different: one was hyper-convergent (vsan) while the other was not
    - One was based on DHCP
    - The name of the centos 7 operating system templates in each vSphere cluster where different as well as other template internal aspects. Aspects such as the user name, password, boot type, etc

From this step we also reported a change request as the user and password in the template is a sensible credential that was required to be supplied somehow. Following we held a meeting with the ICG team and we decided to route these credentials in the same way as other credentials are passed to the IEM. As a result:

(cc) BY-SA
```

- IEM was modified to accept custom secrets
- ICG was modified to consume those credentials from environment variables using a convention
- IEM un-deployment feature was integrated in the test

Next, a meeting was planned to show SIMPA the IaC used and the procedure to deploy in an infrastructure similar to the one they are using. But before the meeting we decided to shift the approach and apply the IaC through the PRC as it was then ready to accept bundles.

Therefore, we introduce PRC in the SIMPA experiment Figure 34. Bundling the IaC was a late decision based on the feedback from the usage of the IDE, and the deployment workflow. The new approach also covered the un-deployment feature of PRC.

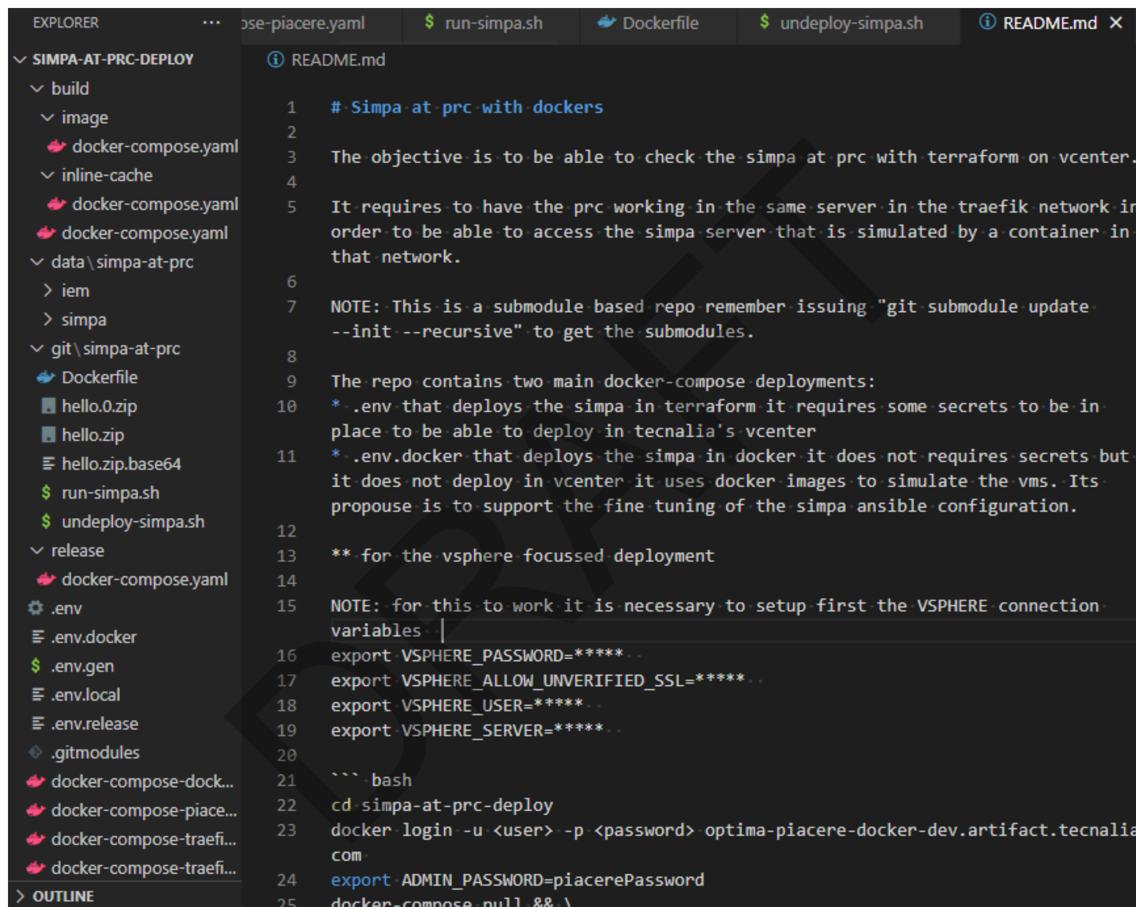


Figure 34 – SIMPA at PRC.

The demo took one hour: we explained to SIMPA the approach followed and we verified that the infrastructure was deployed, the agents were installed, and the information was received.

Following week we held a meeting with SIMPA, in which we updated the Terraform to their on-premise settings. During that deployment, the virtual machines were created but the agents were failing to install. The following week they reported that they adjusted the infrastructure, and the agents were installed as well.

Finally, we are currently experimenting the deployment with Ericsson, the deployment of the infrastructure in their scenario. They are currently experiencing issues with the agents' installation that are being analysed.

5 Lessons learnt and outlook to the future

During the implementation and piloting of the WP6 monitoring components we have find out that:

- A single point of entry is good for use cases, but it adds problems
- For testing a monitoring feature, many requirements must by in sync
- For being able to develop, some elements must be in place for some time
- Automation and ease of use are many times a good investment
- Extensibility is important in DevSecOps

The first lesson learnt is that it was very important to drive all features from the IDE, from the deployment of the monitoring to the access to the different features. On the dark side it involved that for the monitoring to work, it was necessary that IDE, ICG, PRC and IEM were developed and in sync.

Sometimes the introduction of a change in a component requires a chain of changes back to the IDE that take time and affects the ability to test the new features or changes.

In this sense a good approach has been to use the components one by one as they were available and synchronised enough. In order to implement this approach, the Docker and the Docker-compose technology have been an unvaluable resource.

Some of the components of this Work package require some pre-existing assets for being developed. The availability of such assets or "faked versions of them" is critical for supporting the agile development and the consistency in the results.

In this sense the establishments of fake monitoring agents in the infrastructure have been a crucial element for the development of other components such as the self-learning components or the monitoring dashboard. But it was not perfect: there were situations in which, after a reset of the environment, the lack of pre-existing metrics in the time-series database has been a problem. Problem in the sense that we were forced to wait, and problem in the sense that we were not able to replicate the behaviour as the input data was not the same.

Another lesson learnt has been that automation and means to repeat tests fast and consistently, were very useful tools. During the agents test with the IEM and ICG, and especially in the experimentation with SIMPA, we learnt that having ways to do and replicate tests was an important communication mechanism.

For this automation we used Docker, Docker-compose and shell scripting in order to reduce to the minimum the number of steps required to replicate some tests and to make them access to as much information technology profiles as possible.

Another lesson was that providing extensibility mechanism is very important for providing the necessary adaptation features. During this last period, we have introduced the monitoring rules and the Ansible strategy as mechanisms to adapt Monitoring to cover a greater number of situations.

As an outlook to the future, we would like to explore:

- Ways to feed the same time series data
- Knowledge base for monitoring rules
- Complement the availability metric

DRAFT

CC BY-SA

Having ways to feed the same time series data will help in the development and in tracing issues in the monitoring components. For some purposes, such as the PSL development, we have exported ranges of times to csv and used them in the PSL algorithm. It will be nice to find a way to "record" some ranges of time from the time series database in a way that we are able to replicate the injection from the fake agents in the same rate. This will help us to replicate the behaviour activating most of the PIACERE workflows.

The creation of a knowledge base for monitoring rules, will correlate monitoring conditions with strategies that can be applied, and specifically Ansible related playbooks to solve the issues.

During this last period, it has not been possible to have a stable source of metrics from the scenarios as they are continuously improving other aspects of PIACERE such as DOML which requires continuous redeployments. As a mitigation mechanism we have been monitoring the PIACERE integration platform which have provided a stable flow of metrics, allowing to feed the training algorithms. Additional platforms can be monitored to check more complex scenarios looking forward to the exploitation of PIACERE.

DRAFT

CC BY-SA

6 Conclusions

Along this document, we have presented the current state of the development of the PIACERE run-time monitoring and self-learning, self-healing platform, together with the rationale that supports the decisions taken in this period.

As we have stated in the executive summary, the objective in this period has been finalization and application of the Key results to the scenarios. Specifically, **KR11 - Self-learning and self-healing mechanisms** and **KR12 - Runtime security monitoring**.

During this process we have introduced many final changes in the existing components of the covered key results:

- Metrics have been simplified
- Availability has been introduced
- DOML has been extended
- Self-healing strategies have been extended
- Monitoring agents have been integrated with ICG
- Self-healing reports alerts to the catalogue

But the main focus has been the experimentation with the use cases, and the adjustments of the components from that experimentation. Central point of the experimentation has been in the deployment of the agents in different use cases with different architectures.

From that testing we have interacted with many other components providing the necessary feedback to make things work at this level: IEM, PRC and ICG.

The experimentation of the remaining features will be done in the following months in the context of the support of the use cases. The testing will be focussed on the long-term maintenance of the systems, rather than in the exploration of the deployment and un-deployment capabilities.

DRAFT

(cc) BY-SA

Annex A.Implementation, delivery and usage

In this section we describe the implementation delivery and usage details of the different components that build up the WP6 key results KR11 and KR12.

Table 7 – Components and KR relations

Disk	KR11	KR12
Monitoring Controller	X	X
Performance Monitoring	X	
Security Monitoring	X	
Performance Self-learning	X	
Security Monitoring		X
Security Self-learning		X
Self-healing	X	

A.1. Monitoring Controller

A.1.1. Implementation

This subsection is devoted to describing the implementation details of the monitoring controller component. First, we describe the internal architecture and then we describe the technical details.

The main architecture of the component is depicted in the following Figure 35. In this architecture, seven different components can be distinguished: Connexion, Monitoring Controller, and five clients to communicate with the rest of the components of PIACERE monitoring, self-learning and self-healing components.

DRAFT

CC BY-SA

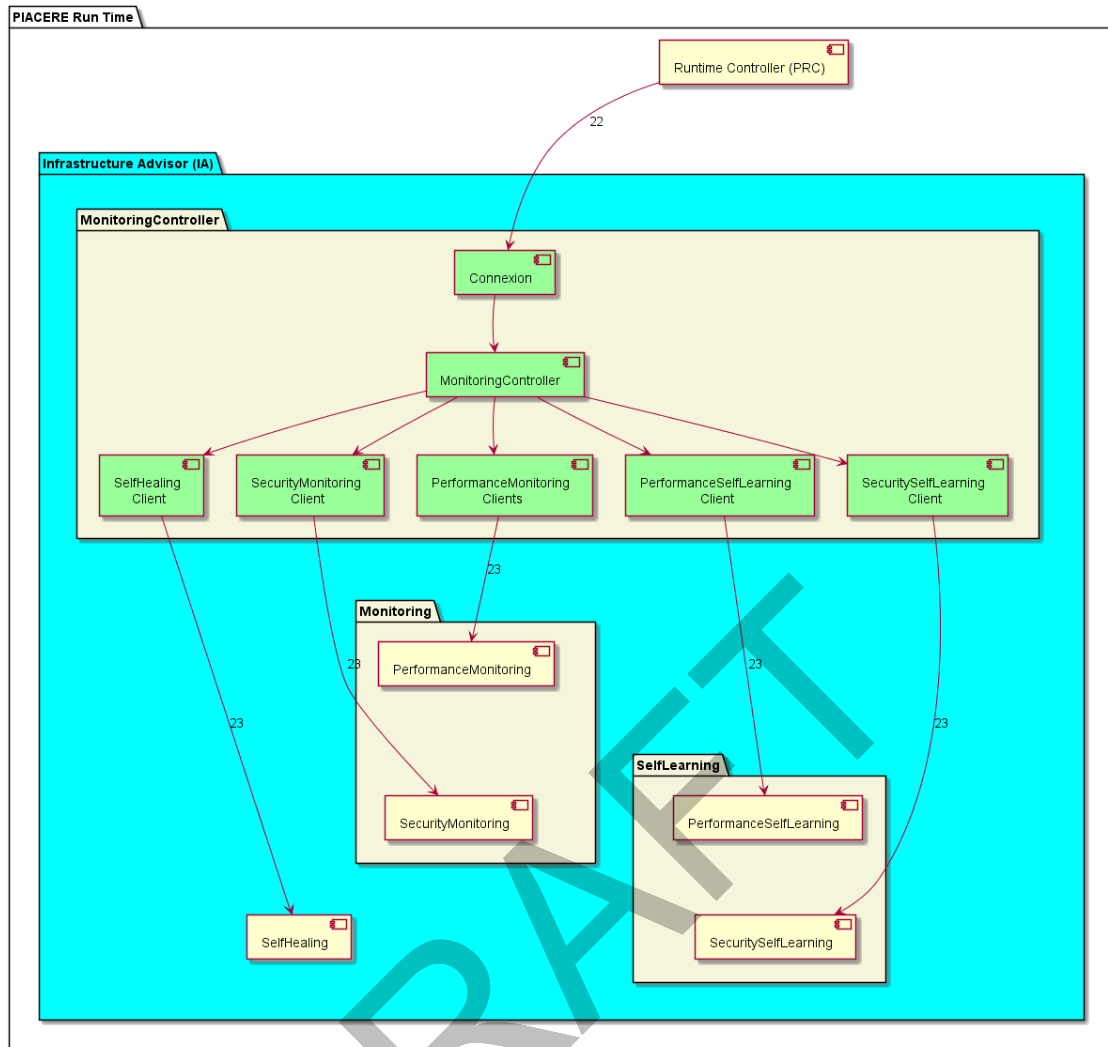


Figure 35 – Monitoring Controller internal architecture.

This final version of the Monitoring Controller makes use of five client components, as self-healing requires to know that a concrete deployment has been activated. We have integrated this fifth component "Self-healing" during the development of the last strategies. During this development we have identified the need to setup refined strategies in the self-healing component for each active deployment.

The Monitoring Controller internal components are the following.

- Connexion: This is an open-source component from Zalando[5] that enables the specification-first approach in Python.
- Monitoring Controller: This is the main component where the forwarding and the configuration are managed.
- Performance Monitoring Client: This is an autogenerated component from the OpenAPI of the performance monitoring with the OpenAPI generator[6].
- Security Monitoring Client: This is also an autogenerated component with the OpenAPI generator.

[5] https://github.com/zalando/connexion
[6] https://github.com/OpenAPITools/openapi-generator

BY-SA

- Performance Self-learning Client: This is also an autogenerated component with the OpenAPI generator.
- Security Self-learning Client: This is also an autogenerated component with the OpenAPI generator.
- Self-healing Client: This is also an autogenerated component with the OpenAPI generator.

This prototype has been developed using Python, which is an interpreted class-based, high level, object-oriented and general-purpose programming language. We have chosen Python as it is easier to read, learn and write and is ideal for the fast implementation of low complexity code as the one we have done in this component.

The component is packaged using Docker technology to simplify the Python requirements and environment management. This is also a requirement for the future integration of PIACERE components into the PIACERE framework.

A.1.2. Delivery

The component code is available at PIACERE code repository at:
https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/mc/-/tree/y3

There are many ways to run this component:

- Run the component in isolation
- Run with Docker
- Run with Docker compose
- Run with Vagrant

Each approach is described into its corresponding README in the PIACERE code repository.

A.1.2.1.Component in isolation

The installation of the component in isolation is described at:
https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/mc/-/tree/y3

The requirement to run the component in isolation is to have Python 3.5.2+. In order to execute the component we have to carry out three steps:

- Download the code
- Install the requirements
- Launch the Python module

To download the code we will use git:

```
git clone https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/mc.git
```

To install the requirements we will use pip, so we will require to have the pip3 Python tool:

```
cd mc
pip3 install -r requirements.txt
```

NOTE: the module has been developed on linux and therefore even if Python is multi-platform, we cannot ensure that the requirements are multiplatform as well. Therefore, running this step in non linux systems may have some issues. In case you have a different operating system, you can proceed as indicated in A.1.2.4.

DRAFT

(cc) BY-SA

To launch the Python module we require to have the port 8080 available and run:

```
python3 -m mc
```

A.1.2.2.Docker

The installation with Docker is also described at:
https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/mc/-/tree/y3

The requirements to run the component with Docker is to have Docker installed, we have used Docker version 20.10.10 with linux/amd64 architecture. In case you have a different operating system, you can proceed as indicated in A.1.2.4. In order to execute the component we have to carry out three steps:

- Download the code
- Build the image
- Run the image

To download the code we will use git:

```
git clone https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/mc.git
```

To build the image:

```
cd mc
docker build -t mc .
```

NOTE: the image relies on linux kernel, therefore it requires a Docker installation able to run linux based machines.

To run the image in a container we require to have the port 8080 available and run:

```
docker run -p 8080:8080 mc
```

A.1.2.3.Docker compose

The installation with docker-compose is described at:
https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pm/-/tree/y3

This docker-compose is a partial integration of components of WP6, currently we cover two components: monitoring controller and performance monitoring; in the future we will cover all the remaining components: (security monitoring, performance self-learning, security self-learning and self-healing.).

The requirements to run the component with Docker is to have Docker installed, we have used Docker version 20.10.10 with linux/amd64 architecture and docker-compose version 1.29.0 . In case you have a different operating system, you can proceed as indicated in A.1.2.4. In order to execute the component, we have to carry out three steps:

- Download the code
- Setup relevant variables
- Build the images
- Run the docker-compose

To download the code we will use git:

```
git clone -recurse-submodules https://git.code.tecnalia.com/piacere/private/t61-runtime-monitoring/pm/pm-deploy.git
```

DRAFT

To setup relevant variables we need to identify the variables without values, and give value to them:

```
echo list variables to be setup
cat .env | grep -e ".*=\s*$"
```

Assign values to those variables. The current set of values are the ones show below, but they are subject to change as the development advances, therefore it is advisable to check the current list using the instruction above (cat …)

```
export SERVER_HOST=192.168.56.1.nip.io
```

NOTE: https://nip.io is a service that allows doing a mapping between any IP to a hostname.

To build the images:

```
cd pm-deploy
docker-compose build
```

NOTE: the image relies on linux kernel, therefore it requires a Docker installation able to run Linux-based machines.

To run the docker-compose we will need the port 443 available:

```
docker-compose up
```

A.1.2.4. Vagrant

In case we do not have a Docker compatible operating system, or we cannot install the Docker desktop version, we will be able to use VirtualBox to instantiate a virtual machine. The easiest way to do so is to use Vagrant.

```
mkdir piacere-vagrant
cd piacere-vagrant
vagrant init  --minimal ubuntu/jammy64
```

edit the file named Vagrantfile with the following content

```
VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/jammy64"
  config.vm.network "forwarded_port", guest: 443, host: 443
end
```

after that we can create the virtual machine and connect to it

```
vagrant up
vagrant ssh
```

inside we proceed to install Docker and Docker Compose
https://docs.docker.com/compose/install/linux

Then we follow the Docker compose delivery method described in the previous section

A.1.2.5. Licensing information

This component is offered under Apache 2.0 license. More detailed information can be found in the GitLab repository.

DRAFT

CC BY-SA

A.1.3. Usage

The Monitoring controller can be used through its REST API, described at: https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/mc/-/blob/y3/src/mc/openapi/openapi.yaml

In order to access that API in the running component, we need to specify the HTTP protocol, the host and the port. Then it will be possible to access the REST API documentation in the same running instance where we can invoke the services.

For the component in isolation, the way to access the swagger UI, showing the REST API, will be http://localhost:8080/api/v1/ui/, in the rest of the execution options the access will depend on the server and the port specified and it will look like https://192.168.56.1.nip.io:8443/mc/api/v1/ui/

In that address we will find the standard swagger UI shown in Figure 36. The swagger UI will list the operations available and it will allow us to invoke them.

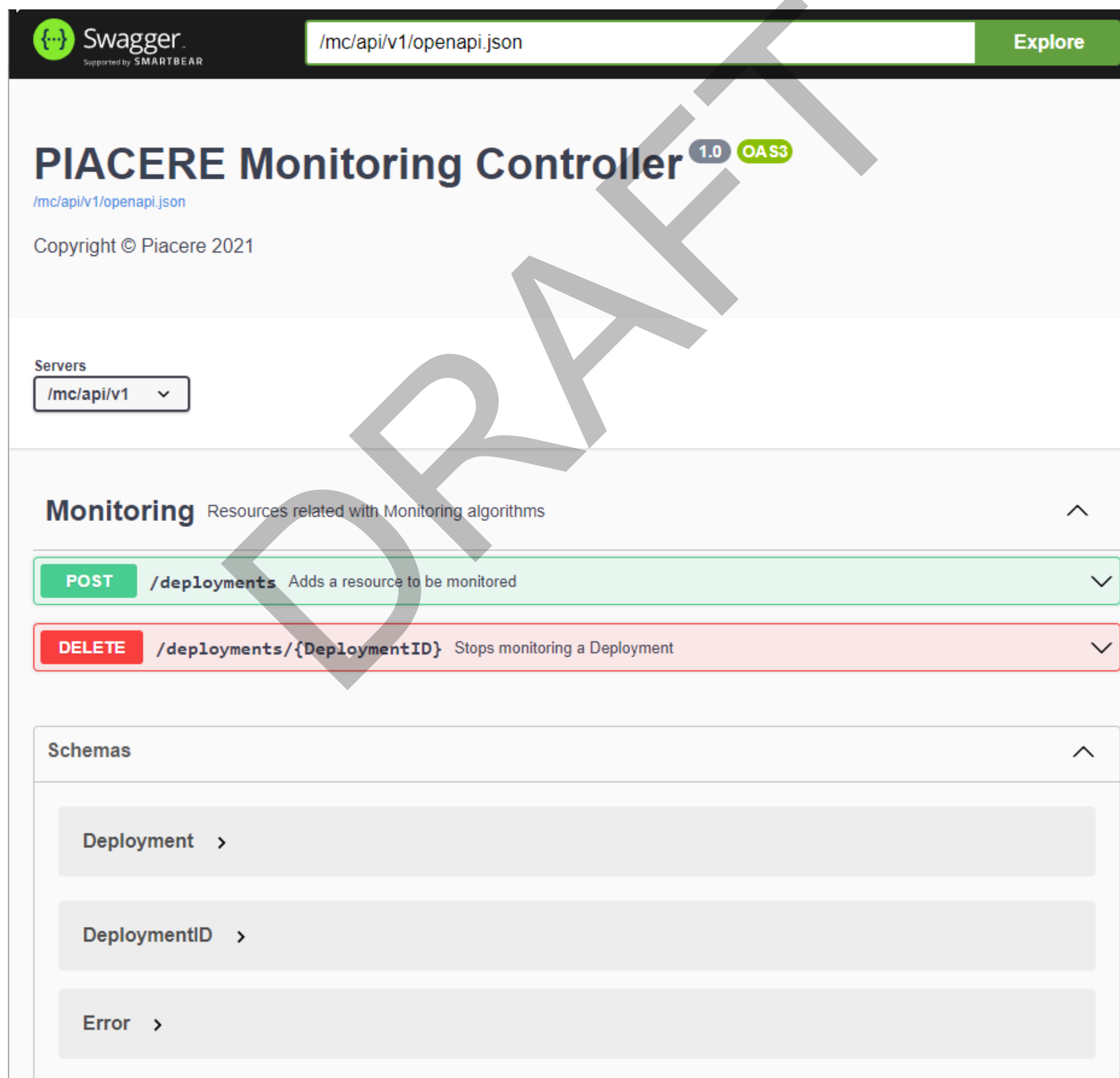


Figure 36 – Monitoring Controller swagger UI.

Anyway, even if the swagger UI provides a valuable resource to understand and test the API, the real way to use the component will be to integrate it with other components. To do so the best way is to get profit from the OpenAPI based client code generators such as:

- Openapi-generator: https://github.com/OpenAPITools/openapi-generator
- Swagger-codegen: https://github.com/swagger-api/swagger-codegen

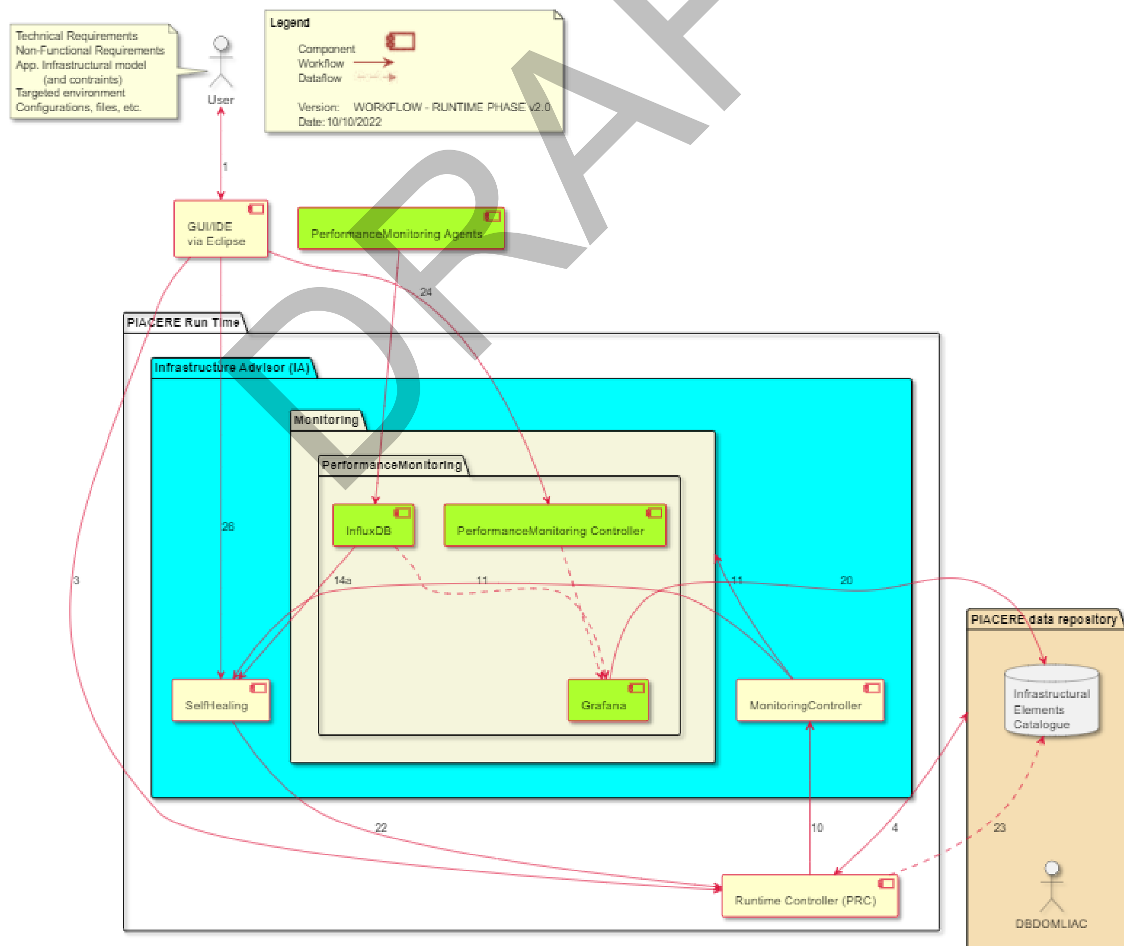
A.2. Performance Monitoring

A.2.1. Implementation

This subsection is devoted to describing the implementation details of this last version of the performance monitoring. First, we describe the internal architecture and then we describe the technical details.

The main architecture of this last version is depicted in the following Figure 37. In this architecture, four different components can be distinguished (highlighted in green): Performance Monitoring Controller, Influxdb, Grafana and Performance Monitoring Agents. The main purpose of these components is described below.

From the previous version we have added links from the IDE to the performance monitoring dashboard to provide the users an easier access to the monitoring information.



DRAFT

Figure 37 – Performance Monitoring last version architecture.

(cc) BY-SA

This last version of the Performance Monitoring is composed by four components, three of them will run together with the PIACERE runtime framework and the other one will run in the deployed infrastructures. The components in the PIACERE runtime framework are:

- Performance Monitoring Controller: this is the main component that receives the start and stop requests by the Monitoring controller and configures Grafana in consequence.
- Influxdb: is a time series database that will receive the information from all the Performance Monitoring agents throughout all the active deployments: This is an open-source component[7] that enables the storage of time series.
- Grafana: is a time series rendering web interface that includes functionalities to keep track of thresholds and sends notifications when the thresholds are exceeded. This is an open-source component[8]

The component running in the deployed infrastructures is the Performance Monitoring agent. The monitoring agent gathers multiple parameters from the runtime infrastructure that run the components of the deployed application. The Performance Monitoring agent is implemented using an open-source component[9].

The Performance Monitoring Controller prototype has been developed using Java, more specifically the Java Spring Boot framework[10] that is an open source, enterprise-level framework for creating standalone, production-grade applications. We have created the application using the OpenAPI generator technology, that starting from the OpenAPI specification is capable to generate a Spring Boot architecture to implement that functionality.

Internally we have developed a client from the recent Grafana OpenAPI https://github.com/grafana/grafana/blob/main/public/api-spec.json. This allows us to easily adapt to higher versions of Grafana in case we need to evolve to bring new features or security patches.

Additional dashboard has been integrated to show self-learning computed data, this has introduced the usage of dashboard folders to aggregate the dashboard of each deployment.

A.2.2. Delivery

The component code is available in the PIACERE code repository at:
https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pm/-/tree/y3.

Besides, a testing oriented agent infrastructure can be deployed using the code available at https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pma/-/tree/y3. This enables the feeding of data into the PIACERE monitoring platform without requiring to perform real deployments that may involve some costs.

This component shares the part of the development environment with the Monitoring controller. In that sense it shares some of their ways to be executed:

- Run with Docker Compose
- Run with Vagrant

[7] https://www.influxdata.com/
[8] https://grafana.com/
[9] https://www.influxdata.com/time-series-platform/telegraf/
[10] https://spring.io/projects/spring-boot

DRAFT

(cc) BY-SA

Besides, as this component is composed by separate running services, it makes no sense to apply some of the execution methods available in the Monitoring controller such as: run the component in isolation or run with Docker. However, focussing in the Performance monitoring controller there can be situations, such as during development, where it can make sense to run this component in isolation. For this specific case we provide specific guidelines.

Each approach is described into its corresponding README in the PIACERE code repository.

A.2.2.1. Performance monitoring controller in isolation

The installation of the component in isolation is described at:
https://git.code.tecnalia.com/piacere/private/t61-runtime-monitoring/pm/pmc

The requirements to run the component in isolation is to have Java and Maven. In order to execute the component we have to carry out two steps:

- Download the code
- Launch the spring boot application

To download the code we will use git:

```
git clone https://git.code.tecnalia.com/piacere/private/t61-runtime-monitoring/pm/pmc.git
```

To run the spring boot application

```
mvn run
```

A.2.2.2. Docker compose

The installation with docker-compose is described at:
https://git.code.tecnalia.com/piacere/private/t61-runtime-monitoring/pm/pm-deploy.

This docker-compose is a partial integration of WP6 components. Currently we cover two components: monitoring controller and performance monitoring; in the future we will cover all the remaining components: (security monitoring, performance self-learning, security self-learning and self-healing.). The installation details are available above in section A.1.2.3.

A.2.2.3. Vagrant

In case we do not have a Docker compatible operating system or we cannot install the Docker desktop version, we will be able to use VirtualBox to instantiate a virtual machine. The easiest way to do so is to use Vagrant.

```
mkdir piacere-vagrant
cd piacere-vagrant
vagrant init  --minimal ubuntu/jammy64
```

edit the file named Vagrantfile with the following content

```
VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/jammy64"
  config.vm.network "forwarded_port", guest: 443, host: 443
end
```

after that we can create the virtual machine and connect to it

```
vagrant up
vagrant ssh
```

DRAFT

CC BY-SA

inside we proceed to install docker and docker-compose
https://docs.docker.com/compose/install/linux

Then we follow the Docker compose delivery method described in the previous section

A.2.2.4. Licensing information

This component is offered under Apache 2.0 license. More detailed information can be found in the GitLab repository.

A.2.3. Usage

This component has three different sub-components. In the following subsections we provide the user manual for each of them.

A.2.3.1. Performance Monitoring controller

The Performance Monitoring controller is used through its REST API, described at: https://git.code.tecnalia.com/piacere/public/the-platform/runtime-monitoring/pm/-/tree/y2/git/pmc/openapi.yaml

In order to access that API in the running component, we need to specify the HTTP protocol, the host and the port. Then it will be possible to access the REST API documentation in the same running instance where we can invoke the services.

For the component in isolation, the way to access the swagger UI, showing the REST API, will be http://localhost:8080/pmc/api/v1/ui/, in the rest of the execution options the access will depend on the server and the port specified and it will look like https://192.168.56.1.nip.io:8443/pmc/api/v1/ui/

In that address we will find the standard swagger UI shown in Figure 38. The swagger UI will list the operations available and it will allow us to invoke them.

DRAFT

(cc) BY-SA

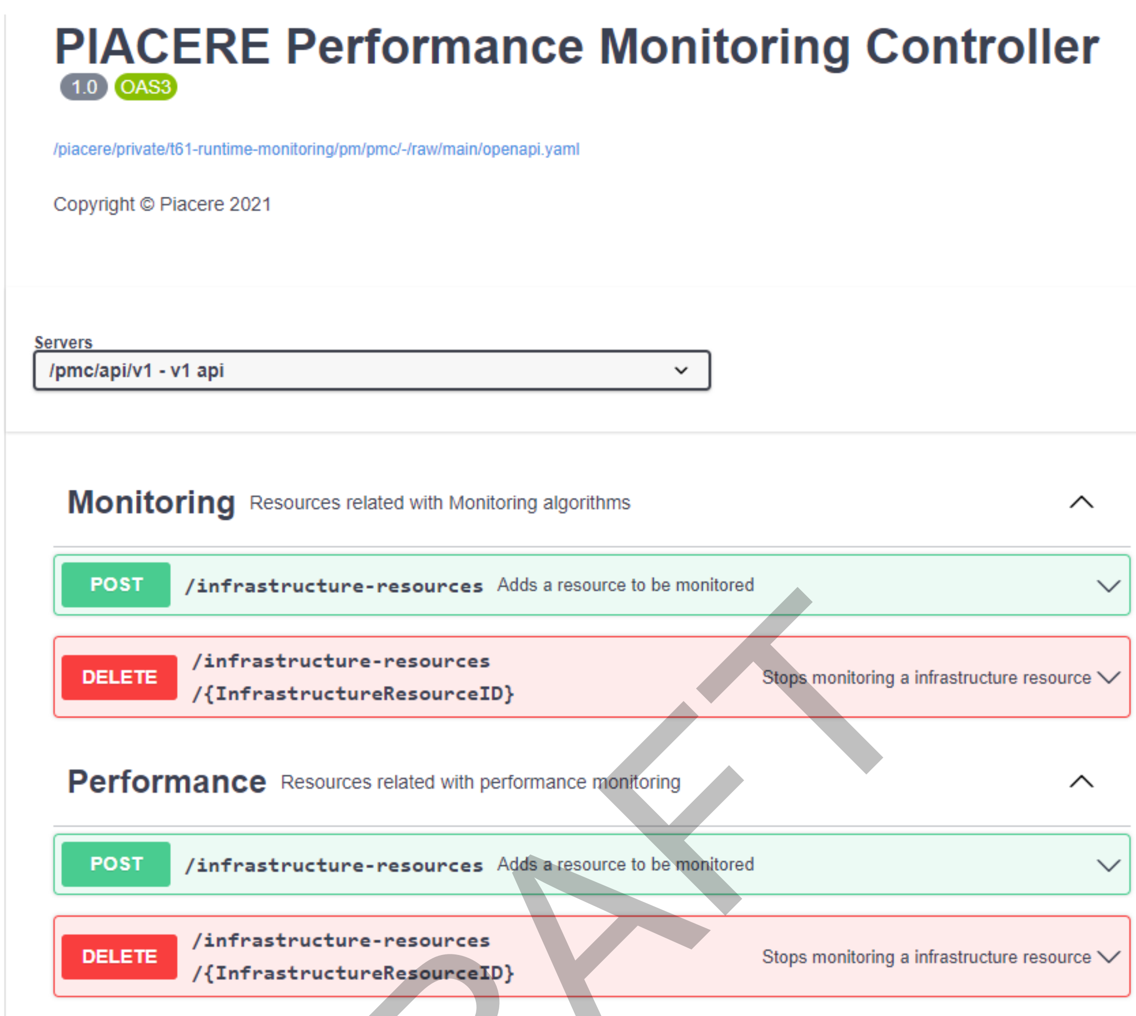


Figure 38 – Performance Monitoring Controller swagger ui.

DRAFT

Anyway, even if the swagger UI provides a valuable resource to understand and test the API, the real way to use the component will be to integrate it with other components. To do so the best way is to get profit from the OpenAPI based client code generators such as:

- Openapi-generator: https://github.com/OpenAPITools/openapi-generator
- Swagger-codegen: https://github.com/swagger-api/swagger-codegen

A.2.3.2.Influxdb

Influxdb will be used following the standard user guideline https://docs.influxdata.com/influxdb/v2.0/. The instance will be available in different URLs depending on the execution method selected. For example, if we use the Vagrant method, it will be accessible at https://influxdb.192.168.56.1.nip.io:8443/. As another example, as shown in the Figure 39, in our internal continuous integration framework the component is accessible at https://influxdb.piacere.esilab.org:8443/

(cc) BY-SA

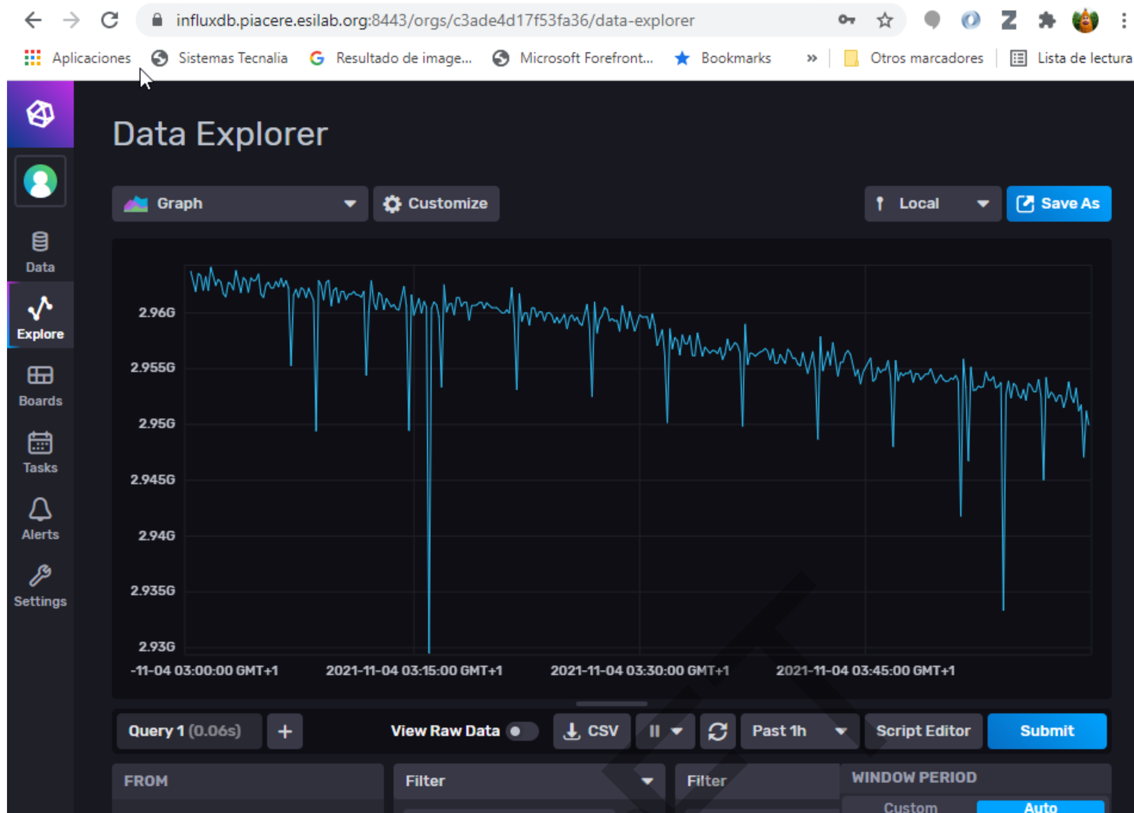


Figure 39 – Influxdb.

A.2.3.3. Grafana

Grafana will be accessible at https://192.168.56.1.nip.io:8443/grafana/

Grafana will be used following the standard user guideline https://grafana.com/docs/grafana/latest/getting-started/getting-started/. The instance will be available in different URLs depending on the execution method selected. For example, if we use the Vagrant method, it will be accessible at https://192.168.56.1.nip.io:8443/grafana, as shown in the Figure 40. As another example, in our internal continuous integration framework, the component is accessible at https://piacere.esilab.org:8443/grafana

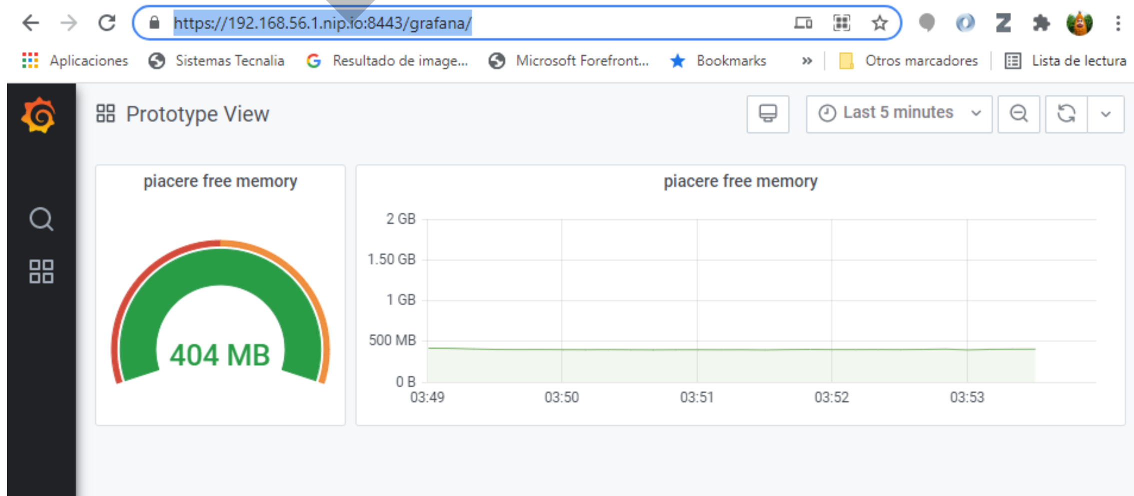


Figure 40 – Grafana

BY-SA

A.3. Security Monitoring

A.3.1. Implementation

This subsection is devoted to describing the implementation details of this final version. First, we describe the internal architecture and then we describe the technical details.

Figure 41 depicts the architecture of the Security Monitoring and Security Self-learning components.

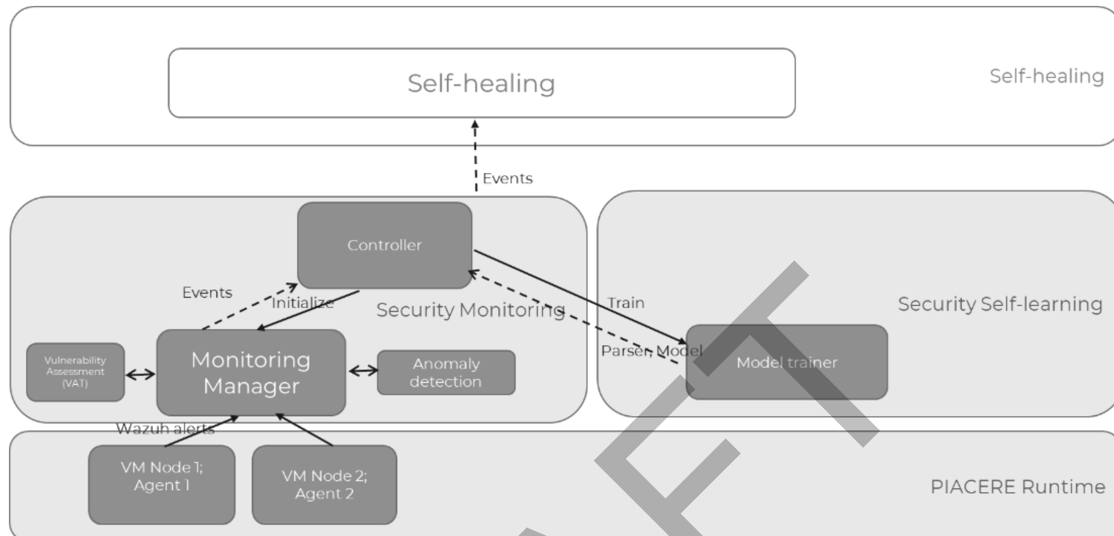


Figure 41 – Architecture of Security Monitoring and Security Self-learning and the integration between these.

Controller exposes underlying APIs of the Monitoring Manager and Model Trainer via RESTful (OpenAPI spec) API. Model Trainer (LOMOS) internally stores trained models in the internal Model Repository (blue component in the upper left of Figure 42). Anomaly Detection component (dark blue component labelled "Python packages") uses the data feed provided by the Monitoring Manager (Celery services in Figure 42) in order to detect anomalies based on the pre-built anomaly detection model built by the Model Trainer (and stored within its internal component – Model Repository).

DRAFT

BY-SA

LOMOS. Services architecture.

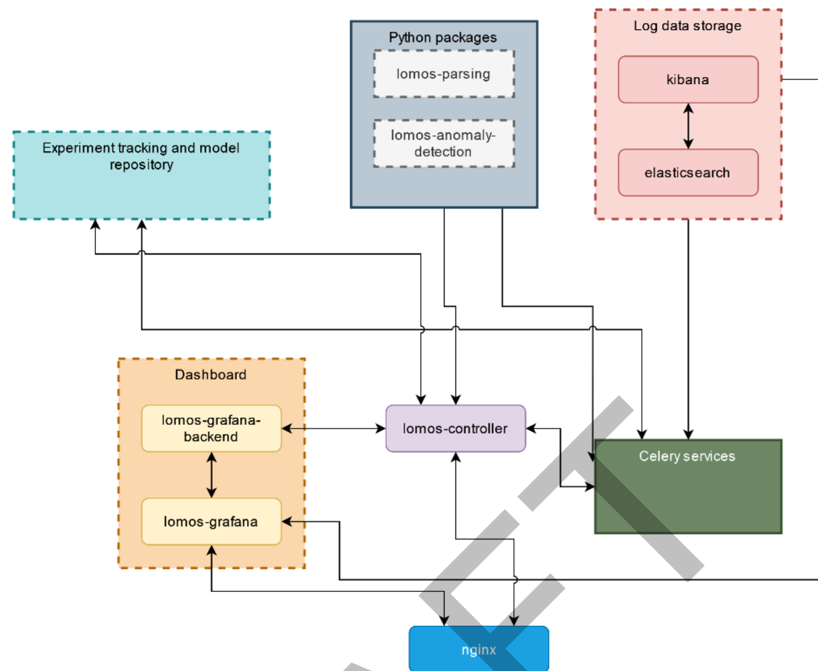


Figure 42 – High-level internals of LOMOS.

Monitoring Manager's Agents residing on the underlying infrastructure provide continuous data feed into the Monitoring Manager's data storage. There are possibilities to extend Monitoring Manager's Agents with other modules such as Vulnerability Assessment Tool (VAT), in order to provide different security-based metrics into the data feed: this is to be considered in the evaluation process.

The Security Monitoring components are developed mainly using Python and Ansible deployment scripts. OpenAPI specification has been developed for the Controller's API (publicly accessible at https://git.code.tecnalia.com/piacere/public/the-platform/runtime-security-monitoring/security-monitoring-controller/-/blob/main/swagger_server/swagger/swagger.yaml).

The Controller uses the Flask framework and its underlying Authentication/Authorization mechanisms. Through the API it provides, it exposes alerts where additional search queries are possible.

Monitoring Manager is developed using deployment of Wazuh 4.2 which already provides agents and ELK stack (based on OpenDistro Elasticsearch) used for storing a plethora of different security metrics. It aggregates and stores alerts stemming from the Agents deployed on the infrastructure. Filebeat deployment is part of these agents. Data stemming from ELK (specifically from Filebeat[11]) is being consumed by the anomaly detection mechanism within the Security Monitoring architecture. Monitoring Manager provides Kibana dashboard so that the user can review all the alerts provided by the Security Monitoring Manager.

DRAFT

[11] https://www.elastic.co/beats/filebeat

CC BY-SA

Vulnerability Assessment Tool (VAT) is the tool developed by XLAB. Its deployment is optional at this point. The planned use of the tool is to provide additional security metrics that could be expressed through NFRs.

A.3.2. Delivery

The Security Monitoring Controller's code is available on the project repository. The Controller provides API endpoints to the Security Self-learning component: https://git.code.tecnalia.com/piacere/private/t64-runtime-security-monitoring/security-monitoring-controller

or the public link:

https://git.code.tecnalia.com/piacere/public/the-platform/runtime-security-monitoring/security-monitoring-controller

To install the Monitoring Controller and Monitoring Manager you should use

https://git.code.tecnalia.com/piacere/private/t64-runtime-security-monitoring/security-monitoring-deployment or on public link:

https://git.code.tecnalia.com/piacere/public/the-platform/runtime-security-monitoring/security-monitoring-deployment

The repository provides deployment scripts you can use (docker-compose scripts for running the whole stack locally).

A.3.2.1. Licensing information

This component is offered under Apache 2.0 license. More detailed information can be found in the GitLab repository.

A.3.3. Usage

A.3.3.1. Configuration of the deployment

All the configuration can be provided with through the deployment process, however, the default configuration is generated by the PIACERE CI/CD process. If needed, these can be manually changed or configured. The main sections of the provided Monitoring Manager's configuration are:

1. **Client Configuration**: This section configures the connection between the agent and the manager. It includes settings for the server address, port, protocol, configuration profile, notification time, reconnect time, and encryption method.
2. **Client Buffer**: This section defines the agent buffer options, including the maximum queue size and the maximum number of events per second that can be sent from the agent to the manager.
3. **Policy Monitoring**: This section controls the behavior of the rootcheck module, which performs system integrity checks. It includes settings for the frequency of rootcheck executions, files and trojans to check, and skipping NFS (Network File System).
4. **Wodles**: This section configures various integration modules:
    - **cis-cat**: Configures the CIS-CAT module for security configuration assessment.
    - **osquery**: Configures the Osquery integration module for querying system information.
    - **syscollector**: Configures the Syscollector module for system inventory, including hardware, operating system, network, packages, and processes.

(cc) BY-SA

5. **SCA**: This section configures the Security Content Automation Protocol (SCAP) module, including the scan frequency and skipping NFS.
6. **File Integrity Monitoring (Syscheck)**: This section controls the behavior of the syscheck module, which performs file integrity monitoring. It includes settings for the frequency of syscheck executions, directories to check, files or directories to ignore, file types to ignore, and skipping NFS, devices, and system files.
7. **Log Analysis (Localfile)**: This section configures log analysis for specific files or commands. It includes settings for the log format, the command to execute or file to monitor, alias, and frequency of checks.
8. **Active Response**: This section controls the behavior of active response actions, which are automated responses triggered by specific events. It includes settings for enabling/disabling active response, certificate authority settings, and response actions.
9. **Logging**: This section specifies the format for internal logs generated by the agent. It includes settings for the log format, which can be "plain," "json," or a combination of both.
10. **Labels**: This section defines custom labels that can be used in the agent's configuration or rules. It includes a key-value pair for a specific label.

A.3.3.2.Monitoring Manager

When the Security Monitoring is deployed you should be able to navigate to the URL of the deployment, e.g.

https://sm-wazuh-kibana.ci.piacere.digital.tecnalia.dev/app/wazuh?security_tenant=global

You should be able to log in to the dashboard provided by the deployment of Security Monitoring services, represented in Figure 43and Figure 44.

DRAFT

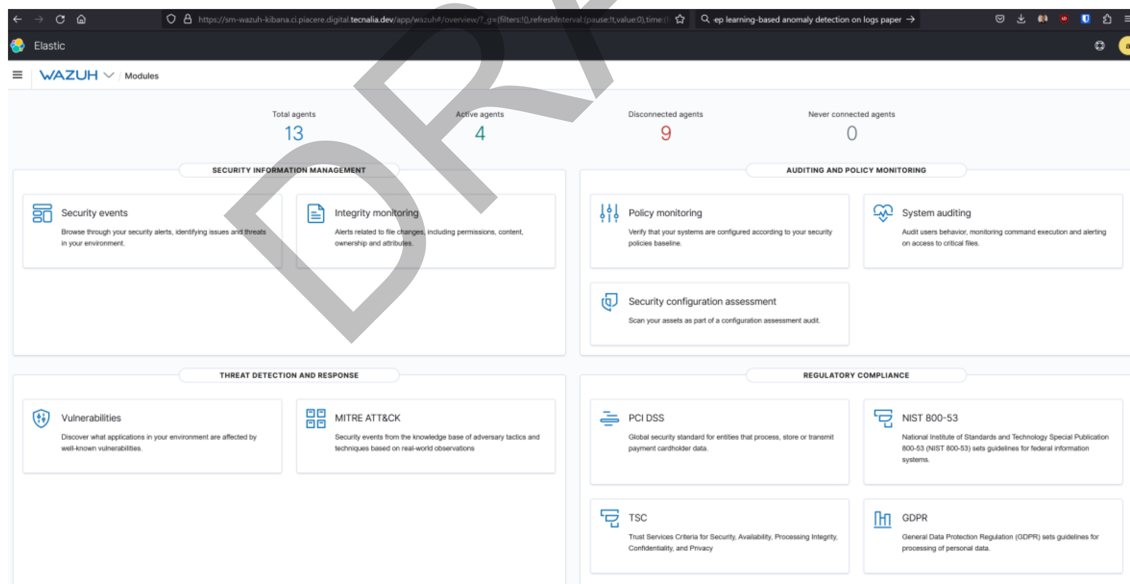


Figure 43 – Default page of the Security Monitoring services.

(cc) BY-SA

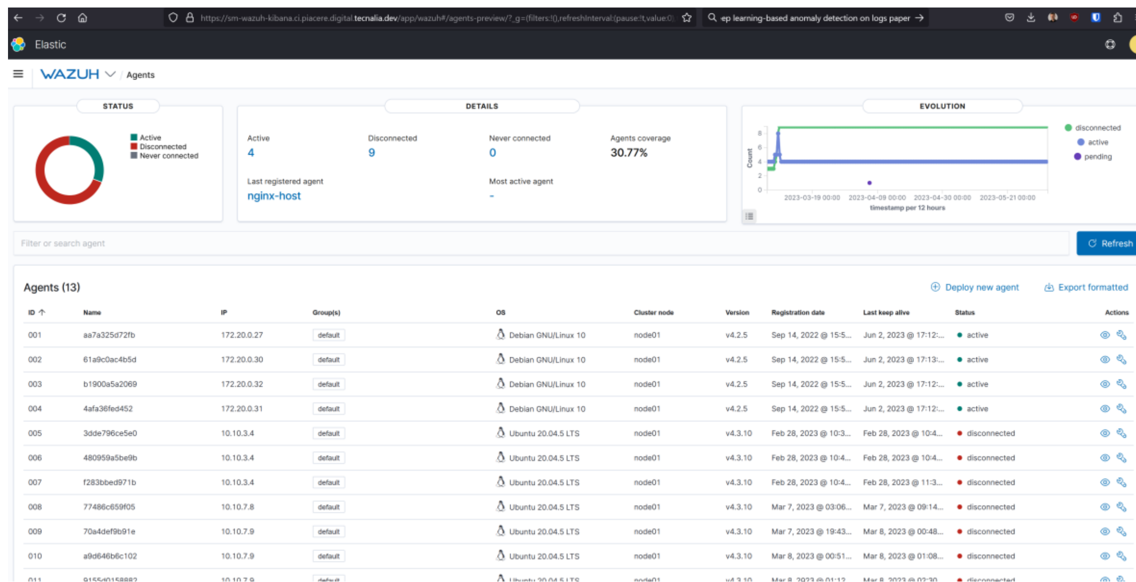


Figure 44 – The list of registered agents with the Wazuh's instance.

Figure above depicts a list of registered agents with the Wazuh's instance provided by Security Monitoring components. Figure 45 depicts Security Monitoring's API as it is served by the Security Monitoring Controller after it is made available (deployed).

Open your browser to here:

https://sm.ci.piacere.digital.tecnalia.dev/security-monitoring/v1/ui/

or, depends on the deployment:

http://localhost:8080/security-monitoring/v1/ui/

Your Swagger definition lives here:

https://sm.ci.piacere.digital.tecnalia.dev/security-monitoring/v1/openapi.json

DRAFT

CC BY-SA

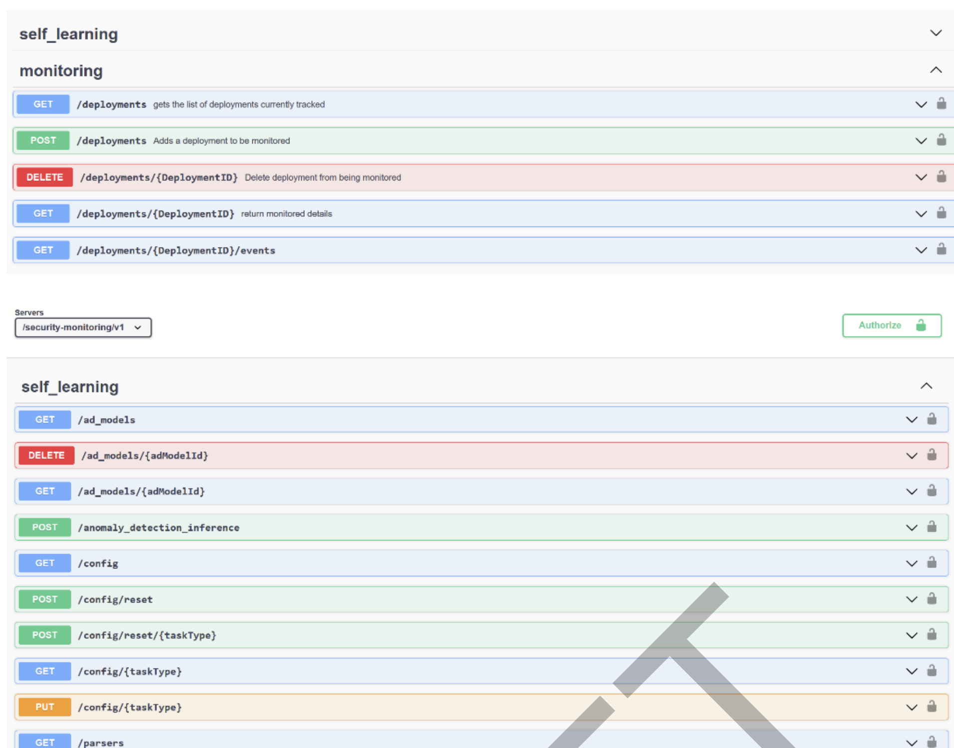


Figure 45 – Security Monitoring part of the Security Monitoring Controller API: the upper figure depicts “monitoring” and the lower figure depicts “self_learnin” part of the API.

Check the running instances:

- Navigate browser to: https://192.168.33.10:5601 (the IP from the inventory file - the inventory file resides in the ansible deployment scripts, this can be reconfigured if needed), login with default credentials (admin:changeme). Navigate to wazuh section on the left hand-side.
- You should see agents registered and running with Wazuh.

In order to check that the service is operative, the user can check whether indices of the ElasticSearch deployment are available and initial listing of entries from the storage’s index can be retrieved. This can be useful for debugging purpose:

- List of indices:

```
curl -X GET https://192.168.33.10:9200/_cat/indices?v -u admin:changeme -k
```

- List all entries in the index wazuh-alerts:

```
$ curl -X GET https://192.168.33.10:9200/wazuh-alerts-4.x-2021.11.03/_search -u
admin:changeme -k
```

A.4. Performance Self-learning

A.4.1. Implementation

This subsection is devoted to describing the implementation details of this final version of the performance self-learning. First, we describe the internal architecture and then we describe the technical details.

DRAFT

BY-SA

The Performance Self-learning component is composed by different solutions and approaches to deal with its goal. In order to achieve its main objective, the Performance Self-learning is composed by different subcomponents portrayed in the following Figure 46.

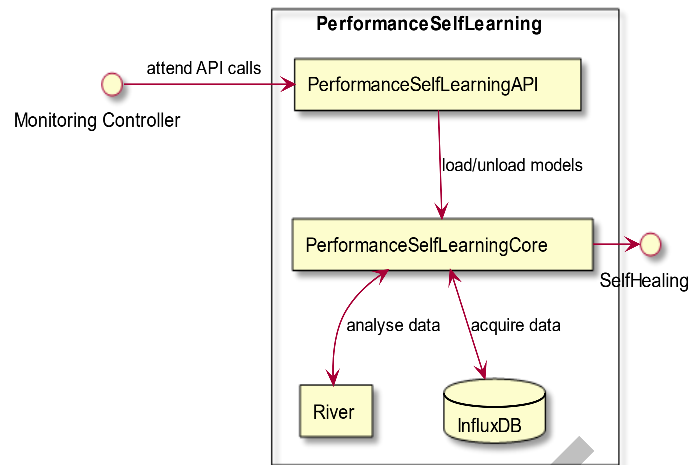


Figure 46 – Architecture of the Self-learning component.

In this architecture, two main different components can be distinguished: PerformanceSelf LearningCore and PerformanceSelfLearningAPI. The PerformanceSelfLearningCore component is also composed by two components: River and InfluxDB.

- PerformanceSelfLearningCore: This component will be in charge of loading or creating Concept Drift and Anomaly Detection learning models whenever a deployment loading has been notified or stopping the learning process on any unloading. This component will also be in charge of feeding the models with data. Finally, once data is analysed, SelfHealing component is notified in case of exceeding the idle threshold for the metric being monitored.
- PerformanceSelfLearningAPI: This component will be in charge of notifying the PerformanceSelfLearningCore component of the loading or unloading of any deployment.

The PerformanceSelfLearningCore is also split in two components:

- River: The library that implements the Concept Drift and Anomaly Detection algorithm
- InfluxDB: The time series database from which the PerformanceSelfLearningCore will receive the data to feed the models.

River[12] is a library for developing online machine learning solutions in Python. It was created by the combination of two of the most popular stream learning packages: scikit-multiflow and creme. Its main innovation is the use of pipelines to transform data in the process of data digestion. It also provides different learning models out of the box, specialized in jobs such as anomaly detection, classification, clustering, regression, etc. The library also offers the Half Space Trees (anomaly detection), Random Forest Regressor (incremental learning) and ADWIN (drift detection) used in the component. River has been the basis for the development of the incremental learning and anomaly detection, and it will also be the basis for the drift detection. After using River with the toy dataset, we have successfully confirmed that it is the suitable library to develop the Self-learning component in the PIACERE project.

[12] https://riverml.xyz/

DRAFT

(cc) BY-SA

For data provision, the official InfluxDB Python library is used, due to the use of InfluxDB as the data storage. Its use is seamlessly integrated in the current implementation, allowing data retrieval for different date ranges providing the ability to use online learning that best fits the component.

This component will also require the integration with different components with a RESTful API to be aware of new deployments to be analysed and to warn the Self-Healing component about any differing behaviour. A Flask server is used to provide the API easing the integration with different components.

Due to the use of Python programming language by the previous libraries, Python has also been selected as the main language of the prototype.

A.4.2. Delivery

The Performance Self Learning's code is available on the project repository:

https://git.code.tecnalia.com/piacere/public/the-platform/self-learning/-/tree/y3

Inside at root level it contains the typical python project plus a Dockerfile and a docker-compose structure.

The structure of the Performance Self-learning inside *src* folder is split in two main sections. The first section, the clients (folder "clients"), contains REST clients' APIs generated with OpenApiGenerator[13].

The second section (folder "psl"), contains the structure of the component. PSL was refactored to a specification first approach where part of the structure has been created with OpenApiGenerator. The relevant folders that contain the logic of PSL, apart from those parts in charge of managing the REST API calls, are:

- Adapters: in includes a generic adapter to get input data from InfluxDB or from CSV (comma-separated values) files in a seamless way. InfluxDB adapter is necessary for the runtime usage while CSV was used to check the process with a repeateable set of data.
- Flows: this folder contains the code that implements the PSL flow: get metrics, store inputs, train or predict, store results.
- Openapi: this folder contains the openapi.yaml that is used to generate the stub of this project.
- Predictor: is the part in which the IA algorithm is implemented
- Repository: is the part that persists the deployments_id to be monitored
- Runners: it implements the background runner that checks the deployments every one minute (configurable)

Besides, in the *psl* folder, there is a *config.yaml* file with the default settings of the PSL: here is where we can adjust many aspects of the behaviour of the PSL.

It follows the same delivery approach of the monitoring controller. There are many ways to run this component:

- Run the component in isolation
- Run with Docker
- Run with Docker compose

[13] https://github.com/OpenAPITools/openapi-generator

DRAFT

CC BY-SA

- Run with Vagrant

Each approach is described into its corresponding README in the PIACERE code repository.

A.4.2.1. Component in isolation

The installation of the component in isolation is described at:
https://git.code.tecnalia.com/piacere/public/the-platform/self-learning/-/tree/y3

The requirement to run the component in isolation is to have Python 3.5.2+. In order to execute the component we have to carry out three steps:

- Download the code
- Install the requirements
- Launch the Python module

To download the code we will use git:

```
git clone https://git.code.tecnalia.com/piacere/public/the-platform/self-learning.git
```

To install the requirements we will use pip, so we will require to have the pip3 Python tool:

```
cd self-learning
pip3 install -r requirements.txt
```

NOTE: the module has been developed on linux and therefore even if Python is multi-platform, we cannot ensure that the requirements are multiplatform as well. Therefore, running this step in non linux systems may have some issues. In case you have a different operating system, you can proceed as indicated in A.1.2.4.

To launch the Python module we require to have the port 8080 available and run:

```
python3 -m psl
```

A.4.2.2. Docker

The installation with Docker is also described at:
https://git.code.tecnalia.com/piacere/public/the-platform/self-learning/-/tree/y3

The requirements to run the component with Docker is to have Docker installed, we have used Docker version 20.10.10 with linux/amd64 architecture. In case you have a different operating system, you can proceed as indicated in A.1.2.4. In order to execute the component we have to carry out three steps:

- Download the code
- Build the image
- Run the image

To download the code we will use git:

```
git clone https://git.code.tecnalia.com/piacere/public/the-platform/self-learning.git
```

To build the image:

```
cd self-learning
docker build -t psl .
```

NOTE: the image relies on linux kernel, therefore it requires a Docker installation able to run linux based machines.

DRAFT

CC BY-SA

To run the image in a container we require to have the port 8080 available and run:

```
docker run -p 8080:8080 psl
```

A.4.2.3. Docker compose

The installation with docker-compose is described at:
https://git.code.tecnalia.com/piacere/public/the-platform/self-learning/-/tree/y3

The requirements to run the component with Docker is to have Docker installed, we have used Docker version 20.10.10 with linux/amd64 architecture and docker-compose version 1.29.0 . In case you have a different operating system, you can proceed as indicated in A.1.2.4. In order to execute the component, we have to carry out three steps:

- Download the code
- Setup relevant variables
- Build the images
- Run the docker-compose

To download the code we will use git:

```
git clone -recurse-submodules https://git.code.tecnalia.com/piacere/public/the-platform/self-learning.git
```

To setup relevant variables we need to identify the variables without values, and give value to them:

```
echo list variables to be setup
cat .env | grep -e ".*=\s*$"
```

Assign values to those variables. The current set of values are the ones show below, but they are subject to change as the development advances, therefore it is advisable to check the current list using the instruction above (cat …)

```
export SERVER_HOST=192.168.56.1.nip.io
```

NOTE: https://nip.io is a service that allows doing a mapping between any IP to a hostname.

To build the images:

```
cd self-learning
docker-compose build
```

NOTE: the image relies on linux kernel, therefore it requires a Docker installation able to run Linux-based machines.

To run the docker-compose we will need the port 443 available:

```
docker-compose up
```

A.4.2.4. Vagrant

In case we do not have a Docker compatible operating system or we cannot install the Docker desktop version, we will be able to use VirtualBox to instantiate a virtual machine. The easiest way to do so is to use Vagrant.

```
mkdir piacere-vagrant
cd piacere-vagrant
vagrant init  --minimal ubuntu/jammy64
```

edit the file named Vagrantfile with the following content

DRAFT

CC BY-SA

```
VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/jammy64"
  config.vm.network "forwarded_port", guest: 443, host: 443
end
```

after that we can create the virtual machine and connect to it

```
vagrant up
vagrant ssh
```

inside we proceed to install Docker and Docker Compose
https://docs.docker.com/compose/install/linux

Then we follow the Docker compose delivery method described in the previous section

A.4.2.5.Licensing information

This component is offered under Apache 2.0 license. More detailed information can be found in the GitLab repository.

A.4.3. Usage

The Performance Self-learning component behaviour expects notifications through the RESTful API. The specification of the API can be found at:

```
https://git.code.tecnalia.com/piacere/public/the-platform/self-learning/-
/blob/y3/src/psl/openapi/openapi.yaml
```

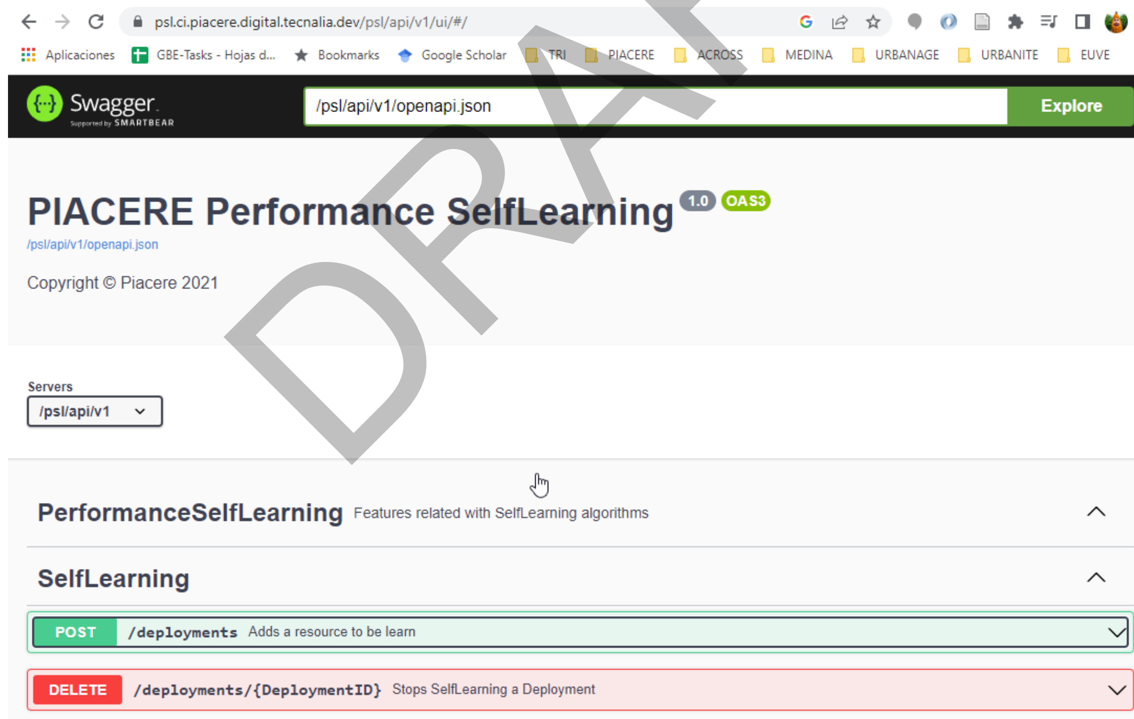


Figure 47 – Performance Self-learning OpenAPI.

A.5. Security Self-learning

A.5.1. Implementation

The Security Self Learning component consists of a single architecture element, referred to as Model Trainer. Its architectural integration with the Security Monitoring is depicted in Figure 41.

BY-SA

The Security Monitoring controller triggers via API the model training, providing the necessary data and configuration files. As a result of the training process, a new log parser based on Drain method, and a new anomaly detection model based on LogBERT are created. These objects belong to the Model Trainer component (Figure 41) and are accessible via API as well.

Additionally, a dashboard is available as a submodule of the existing UI to interact with the Model Trainer both for the training of the log parsers and anomaly detection models, as well as for visualization of intermediate and final results.

Input:

- Data stemming from the Security Monitoring component. The data is already aggregated from different sources by the Security Monitoring component using ELK, which is directly accessed by the Model Trainer.
- Security Self-learning component uses dedicated Elasticsearch's indexes that are considered as input for the anomaly detection process.

Programming languages/tools:

- Python: popular data science and machine learning libraries are used, mainly numpy[14], pandas[15], pytorch[16] and transformers[17].

Dependencies:

- Grafana dashboard (deployment).
- ELK stack: storing raw log data.

A.5.2. Delivery

Security Self-Learning is a proprietary component, not available for delivery within PIACERE.

A.5.2.1. Licensing information

This component is a proprietary component.

A.5.3. Usage

As described earlier in this document, the interaction with the Security Self-Learning service will be done exclusively by the Security Monitoring controller, which exposes an API (included below in Figure 48) for such purposes.

Example of reading all available ad_models trained by the self_learning instance that are available to be used by the Security Monitoring Anomaly Detector:

```
curl -X 'GET' 'https://localhost:8080/security-monitoring/v1/ad_models' -H 'accept: application/json'
```

Result (returning object with parent train_id reference and other details of the model:

```
[
  {
    "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
    "train_id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
    "name": "string",
    "description": "string",
```

[14] https://numpy.org/
[15] https://pandas.pydata.org/
[16] https://pytorch.org/
[17] https://huggingface.co/docs/transformers/index

DRAFT

BY-SA

```
    "configuration": "string"
  }
]
```


self_learning

Method	Path	Operation
GET	/config	get_all_task_config
GET	/config/{taskType}	get_task_config_by_name
PUT	/config/{taskType}	update_task_config_by_name
POST	/config/reset	reset_all_task_config
POST	/config/reset/{taskType}	reset_task_config_by_name
GET	/parsers	get_trained_parsers
GET	/parsers/{parserId}	get_trained_parser
DELETE	/parsers/{parserId}	delete_trained_parser
GET	/ad_models	get_trained_ad_models
GET	/ad_models/{adModelId}	get_trained_ad_model
DELETE	/ad_models/{adModelId}	delete_trained_ad_model
POST	/train_log_parser	post_trigger_task_train_log_parser
POST	/train_anomaly_detection_model	post_trigger_task_anomaly_detection
POST	/anomaly_detection_inference	post_trigger_task
GET	/status/{taskId}	get_task_status
POST	/periodic/anomaly_detection_inference	post_trigger_periodic_task
DELETE	/periodic/anomaly_detection_inference/{taskId}	delete_periodic_task
GET	/periodic	get_all_periodic_task
GET	/periodic/status/{taskId}	get_task_periodic_status

DRAFT

Figure 48 – Self-learning API provided by Security Controller.

The dashboard is based on Grafana (see Figure 49 and Figure 50) and is provided as an informative tool that would provide insights on the intermediate steps and results of the training process.

The Security Self-Learning service is expected to be deployed as a single microservice that will be exposed to the Security Monitoring Controller and will coordinate the whole training process, from the connection to the data source containing raw logs, to the training of the log parser and AD model and their storage in the Model Repository

(cc) BY-SA

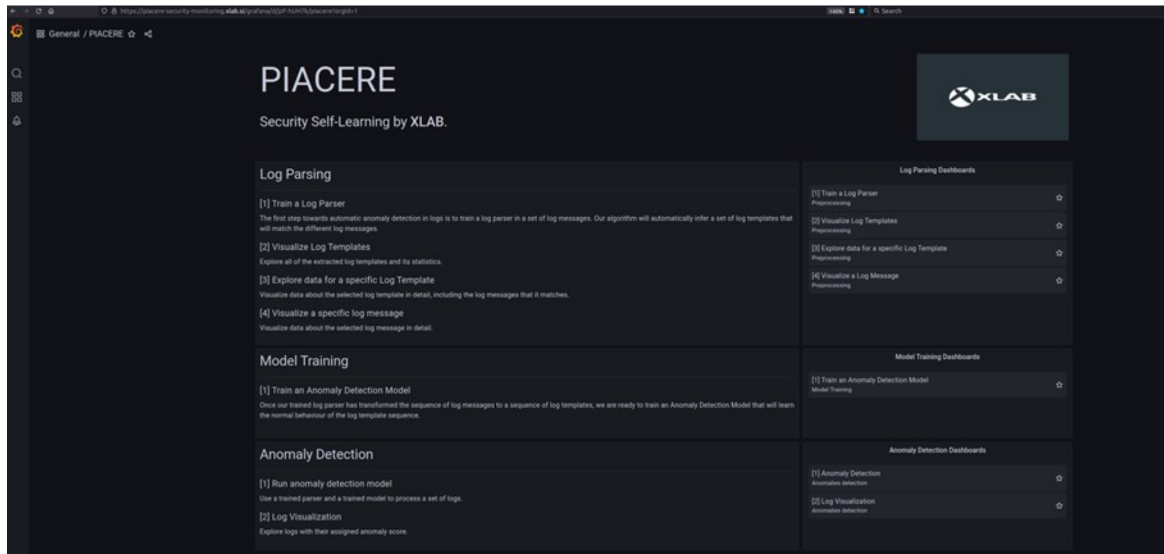


Figure 49 – Dashboard of the Security Self-learning component: main page.

All the different services composing the Security Self-Learning are currently running as standalone services which have to be manually executed. In all cases, conda[18] environments are used to handle dependencies in an isolated manner. The codebase is closed and hosted on private repositories.

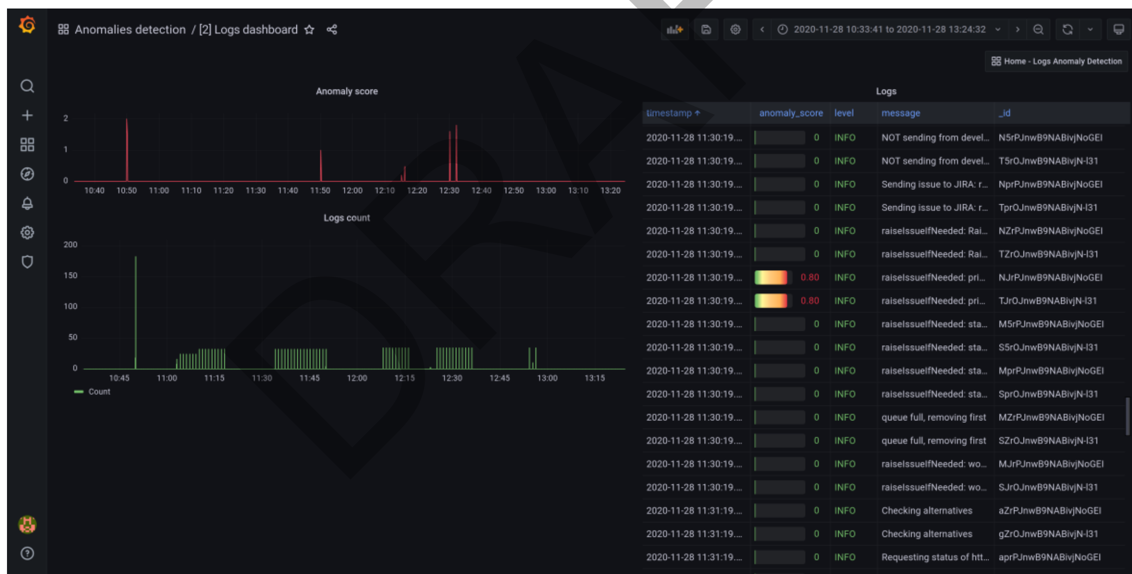


Figure 50 – Dashboard of the Security Self-learning component: inspection of the anomalies detected.

A.6. Self-healing

A.6.1. Implementation

This subsection is devoted to describing the implementation details of this last version. First, we describe the internal architecture and then we describe the technical details.

[18] https://docs.conda.io/en/latest/

DRAFT

CC BY-SA

Self-healing architecture is based on a microservices style which splits the front-end, only for testing purposes in this stage, and the backend, so that´s it´s easier to scale and survive infrastructure issues.

In order to manage the events-oriented architecture, Kafka streaming solution has been chosen.

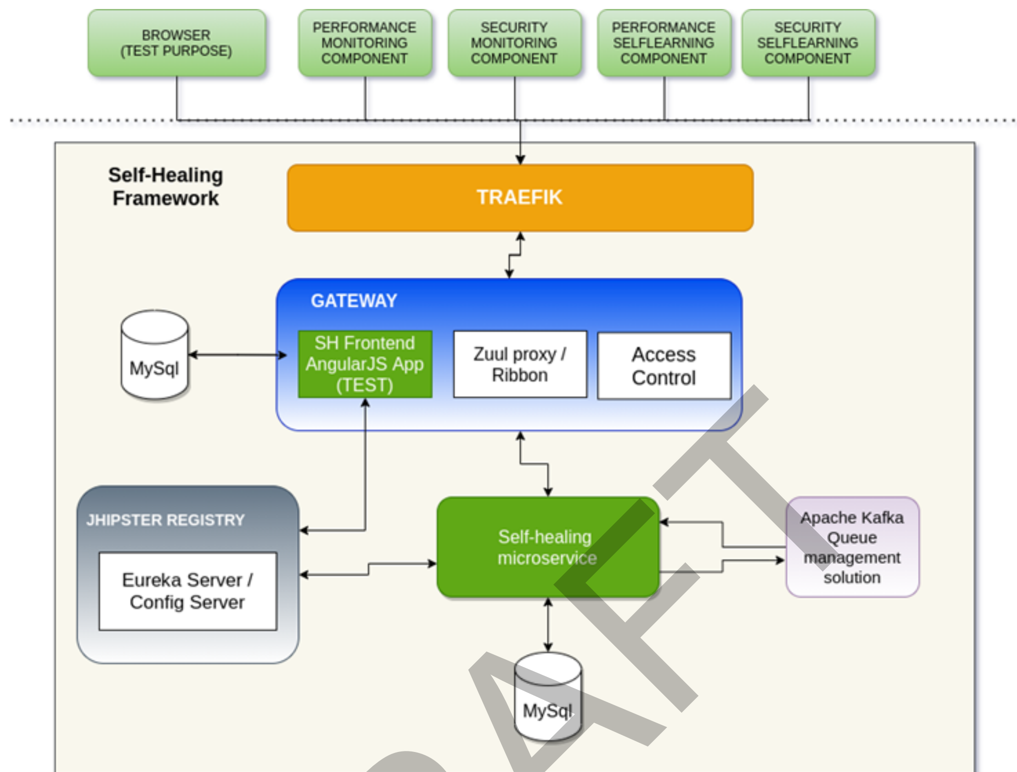


Figure 51 – Self-healing internal architecture.

This last version of the Self-healing is composed by two principal components, which main purpose are briefly described as follows.

- Self-healing service: this component manages all the logic of the self-healing. It implements the necessary logic to treat the notifications received by the components involved in the self-healing mechanism, exposing a REST service in order to simplify the interaction, and the work for communicating with the runtime controller in order to propose the self-healing mechanisms.
- Self-healing test frontend: This component has been developed to simplify the integration and test of the self-healing with the rest of the components.

In addition, some considerations about the other components of the self-healing:

- Access control. JSON Web Token (JWT)[19] mechanism is used. A stateless security mechanism which uses a secure token that holds the user´s login name and authorities.
- Data persistence in MySQL database.
- JHipster Registry[20]. Service discovery using Netflix Eureka[21].

[19] https://jwt.io/introduction
[20] https://www.jhipster.tech/
[21] https://spring.io/projects/spring-cloud-netflix

CC BY-SA
DRAFT

- Apache Kafka[22]: Event streaming solution to capture real-time data from the related components which need to send notifications to the Self-healing component.

This prototype has been developed using JHipster Framework, which provides all the needed technologies and configuration options for a modern web application and microservice architecture.

This framework uses Spring Boot to develop, deploy and test the application. In the client side, the test frontend gateway uses Yeoman, Webpack, Angular and Bootstrap technologies. In the server side the Self-healing microservice uses Maven, Spring, Spring MVC Rest, Spring Data JPA and Netflix OSS.

The technology used to manage the events received is Kafka.

The main structure of the prototype developed in this third stage of the project is composed by the packages shown in the following Figure 52.

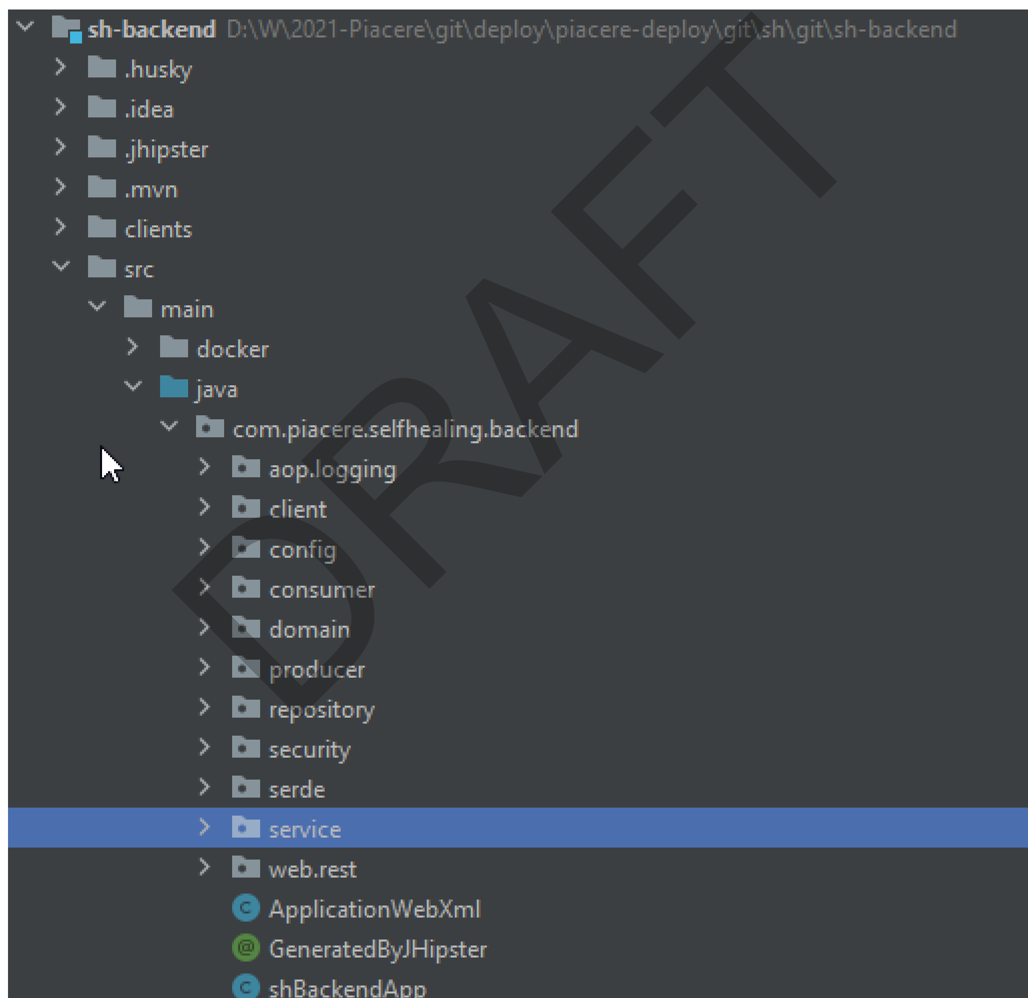


Figure 52 – Self-healing project structure.

[22] https://kafka.apache.org/

(cc) BY-SA

Each of these packages has its own objective and its context within the whole prototype. Furthermore, these packages are also comprised by several JAVA classes. With all this, the main purpose and composition of each component is as follows:

- *com.piacere.selfhealing.backend.aop.logging*: this package is composed by LoggingAspect.java, which defines the aspect for logging execution of service and repository Spring components.
- *com.piacere.selfhealing.backend.client*: Composed by UserFeignClientInterceptor.java which implements RequestInterceptor.java. This class checks and add JWT token to the request header.
- *com.piacere.selfhealing.backend.config*: this package contains all classes related to configuration purposes.
- *com.piacere.selfhealing.backend.consumer*: this package contains classes to consume messages from the queue configured.
- *com.piacere.selfhealing.backend.domain*: this package contains data model classes.
- *com.piacere.selfhealing.backend.domain.enumeration*: this package contains enum objects.
- *com.piacere.selfhealing.backend.producer*: this package contains classes to produce messages to the queue configured.
- *com.piacere.selfhealing.backend.repository*: this package contains Spring Data SQL repository classes.
- *com.piacere.selfhealing.backend.security*: this package contains Spring Security related classes for security management.
- *com.piacere.selfhealing.backend.security.jwt*: this package contains Java Web Token security configuration related classes.
- *com.piacere.selfhealing.backend.serde*: this package contains classes to serialize/deserialize queue messages received.
- *com.piacere.selfhealing.backend.service*: this package contains self healing services for CRUD operations and other requirements needed.
- *com.piacere.selfhealing.backend.service.dto*: this package contains self healing data transfer objects.
- *com.piacere.selfhealing.backend.service.mapper*: this package contains mapping classes to map data transfer objects.
- *com.piacere.selfhealing.backend.web.rest*: this package contains classes to expose Self-healing rest end points.
- *com.piacere.selfhealing.backend.web.rest.errors*: this package contains error classes used in the rest end points.

In the context of Self-healing component, it has been implemented the necessary logic to manage the notifications received with the Kafka streaming solution.

The configuration needed can be found in data/jhipster-registry/central-config/sh/shBackend.yml:

DRAFT

BY-SA

```yaml
kafka:
  bootstrap.servers: kafka:9092
  polling.timeout: 10000
  consumer:
    selfHealingMessage:
      enabled: true
      '[key.deserializer]': org.apache.kafka.common.serialization.StringDeserializer
      '[value.deserializer]': com.piacere.selfhealing.service.serde.SelfHealingMessageDeserializer
      '[group.id]': iec-self-healing
      '[auto.offset.reset]': earliest
  producer:
    selfHealingMessage:
      enabled: true
      '[key.serializer]': org.apache.kafka.common.serialization.StringSerializer
      '[value.serializer]': com.piacere.selfhealing.service.serde.SelfHealingMessageSerializer
  topic:
    selfHealingMessage: queuing.iec_self_healing.self_healing_message
```

Figure 53 – Self-healing configuration.

- Topic: Topic *queuing.iec_self_healing.self_healing_message* has been defined to associate all the events related to the Self-healing logic.
- In the context of Kafka, we need one producer and one consumer to manage the events received
    - Producer: In charge of sending messages to the topic defined Figure 54.
    - Consumer: In charge of processing the messages asynchronously Figure 55.

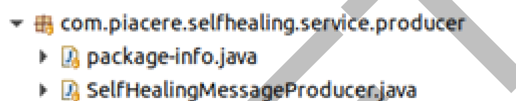


Figure 54 – Self-healing producer.

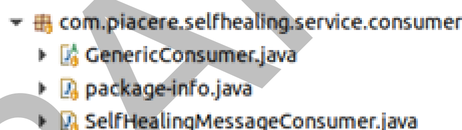


Figure 55 – Self-healing consumer.

A.6.2. Delivery

The code is available in Tecnalia GitLab repository:

https://git.code.tecnalia.com/piacere/public/the-platform/self-healing/-/tree/y3

There are many ways to run this component:

- Run the components in isolation
- Run with Docker compose
- Run with Vagrant

Each approach is described into its corresponding README in the PIACERE code repository.

A.6.2.1. Self-healing in isolation

The installation of the component in isolation is described at:
https://git.code.tecnalia.com/piacere/public/the-platform/self-learning

The requirements to run the component in isolation is to have Java and Maven. In order to execute the component we have to carry out several steps:

- Download the code
- Prepare mysql

DRAFT

(cc) BY-SA

- Start jhipster
- Start backend
- Start frontend

```
git clone https://git.code.tecnalia.com/piacere/public/the-platform/self-healing.git
cd self-healing
echo start mysqld
cd git\jhipster-registry\
mvnw -Pdev,webapp,api-docs -Dskip-tests
cd ..\..
cd git\sh-backend\
mvnw -Pdev,webapp,api-docs -Dskip-tests
cd ..\..
cd git\sh-gateway\
mvnw -Pdev,webapp,api-docs -Dskip-tests
```

Frontends Available services after initialization:

- JHipster registry: http://localhost:8761
- Self-healing test web app: http://localhost:8080
- Self-healing Api Documentation: http://localhost:8080/services/selfhealingservice/v3/api-docs

A.6.2.2. Docker compose

The installation with docker-compose is described at: https://git.code.tecnalia.com/piacere/public/the-platform/self-healing

```
git clone https://git.code.tecnalia.com/piacere/public/the-platform/self-healing.git
cd self-healing
docker-compose up -d
```

A.6.2.3. Vagrant

In case we do not have a Docker compatible operating system or we cannot install the Docker desktop version, we will be able to use VirtualBox to instantiate a virtual machine. The easiest way to do so is to use Vagrant.

```
mkdir piacere-vagrant
cd piacere-vagrant
vagrant init  --minimal ubuntu/jammy64
```

edit the file named Vagrantfile with the following content

```
VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/jammy64"
  config.vm.network "forwarded_port", guest: 443, host: 443
end
```

after that we can create the virtual machine and connect to it

```
vagrant up
vagrant ssh
```

inside we proceed to install docker and docker-compose https://docs.docker.com/compose/install/linux

Then we follow the Docker compose delivery method described in the previous section

A.6.2.4. Licensing information

This component is offered under Apache 2.0 license. More detailed information can be found in the GitLab repository.

DRAFT

(cc) BY-SA

A.6.3. Usage

To test the self-healing functionalities:

- Login to the web app with user/password: admin/admin
- In the administration menu, access openApi.
- Choose SelfHealingService.
- Post a message through the Self-healing notify rest service.
- In this web app, entities menu, can be seen the message received and its status Figure 56.

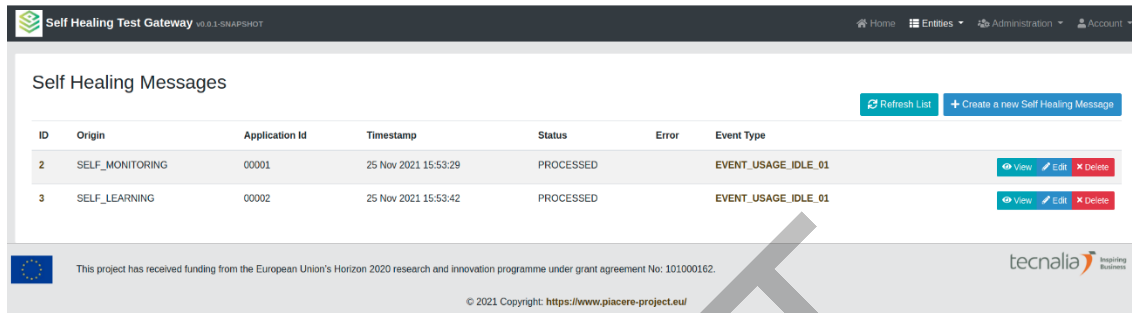


Figure 56 – Messages received in the Self-Healing component.

DRAFT

CC BY-SA