



RAPPORTI TECNICI INGV

INGV RUM
A Lightweight Rule Manager Framework
by INGV
General overview



ISTITUTO NAZIONALE DI GEOFISICA E VULCANOLOGIA

508

Direttore Responsabile

Daniela VERSACE

Editor in Chief

Milena MORETTI

Editorial Board

Laura ALFONSI

Sofia BARANELLO

Carla BARNABA

Christian BIGNAMI

Simona CARANNANTE

Maria Grazia CIACCIO

Luca COCCHI

Luigi CUCCI

Lorenzo CUGLIARI

Elisabetta D'ANASTASIO

Alessia DI CAPRIO

Roberto DI MARTINO

Domenico DI MAURO

Domenico DORONZO

Filippo GRECO

Alessandro IAROCCI

Mario MATTIA

Daniele MELINI

Anna NARDI

Lucia NARDONE

Marco OLIVIERI

Arianna PESCI

Pierangelo ROMANO

Paola SBARRA

Davide SCAFIDI

Maurizio SOLDANI

Sara STOPPONI

Umberto TAMMARO

David ZULIANI

REGISTRAZIONE AL TRIBUNALE DI ROMA N.173 | 2014, 23 LUGLIO

© INGV Istituto Nazionale di Geofisica e Vulcanologia

Rappresentante legale: Presidente INGV

Sede: Via di Vigna Murata, 605 | Roma

Tutti i diritti riservati

www.ingv.it



ISTITUTO NAZIONALE DI GEOFISICA E VULCANOLOGIA

RAPPORTI TECNICI INGV

INGV RUM

A Lightweight Rule Manager Framework
by INGV

General overview

Massimo Fares*, Ivano Carluccio, Peter Danecek, Emiliano Della Bina, Diego Franceschi,
Alfonso Giovanni Mandiello, Michele Maniscalco, Salvatore Mazza, Stefano Pintore, Flavia Tavani

INGV | Istituto Nazionale di Geofisica e Vulcanologia, Osservatorio Nazionale Terremoti, Italy

**Corresponding author*

Accepted 17 October 2025 | *Accettato* 17 ottobre 2025

How to cite | *Come citare* Fares, M., Carluccio, I., Danecek, P., Della Bina, E., Franceschi, D., Mandiello, A.G., Maniscalco, M., Mazza, S., Pintore, S., Tavani, F. (2025). INGV RUM. A Lightweight Rule Manager Framework by INGV. General overview. *Rapporti Tecnici INGV*, 508, 34 pp.
<https://doi.org/10.13127/rpt/508>

508

INDEX

Abstract	7
1. Background	7
1.1 What is a Rule Manager?	7
1.2 How Does a Rule Manager Work?	8
1.3 Applications and advantages of Rule Managers	8
1.4 Technologies and Tools	9
2. INGV Rule Manager: RUM	10
2.1 RUM: Definition and General Overview	10
2.2 Theoretical Foundation	11
2.3 In practice	12
2.4 Technologies Adopted	12
3. Definitions in RUM	13
3.1 Policy	13
3.2 Rules	13
3.3 Actions	13
4. How RUM Works	14
4.1 Rule Engine Layer	15
4.2 Application Layer	16
4.3 Global Objects of the RUM Framework	17
5. RUM Workflow Configuration	18
5.1 Policy Configuration	18
5.2 Rule Configuration	19
5.3 Action Configuration	21
6. Workflow Control in RUM	22
6.1 The Role of SESSION in Workflow Management	23
6.1.1 Example of Flow Control in an Action	23
6.2 Use of Sessions	24
6.2.2 Important Note on the Use of the GOTO Key in Workflow Management	25
7. Installing the RUM Framework	25
7.1 Getting the Framework (Engine Level)	25
7.2 Getting the Project (Application Layer)	25
7.3 Installation with Docker	26
7.3.1 Steps to build and run the Docker image	26
7.3.2 Advantages of using Docker	26
7.4 Expanded Installation	27
7.4.1 Local Installation without Docker	27

7.5 Next Steps	27
7.6 Command-Line Arguments Overview	28
7.6.1 Optional Arguments	28
7.6.2 Usage Examples	29

8. Final considerations	30
--------------------------------	-----------

Acknowledgments	30
------------------------	-----------

Bibliography	30
---------------------	-----------

Abstract

Rule Managers are essential tools for automating decisions, ensuring compliance with regulations and standards, and improving efficiency in data management. This project focuses on the implementation of a simple and lightweight Rule Manager, named *Lightweight Rule Manager Framework by INGV* (or simply RUM), designed to meet the needs of a data center with specific Data Management requirements and an emphasis on data fairness.

The primary objective of RUM is to provide a flexible solution for executing predefined actions on data, ensuring that each dataset is processed according to standardized rules. The system allows the configuration of customized sequences of actions, which can be executed both on a scheduled basis (e.g., via crontab or triggered from the shell) and in response to specific events (e.g., monitoring file insertion in a directory or responding to an endpoint), ensuring wide adaptability.

The INGV RUM is designed to optimize resources and enhance process efficiency by taking into account only data that meet user-defined criteria, such as file directory location or metadata values. Beyond its role in data management, the framework is also suitable for classic Rule Manager applications, proving to be a versatile and effective solution for simplifying data handling while enhancing security and data integrity.

Keywords Policy Management; Rule Manager; Business Automation

1. Background

1.1 What is a Rule Manager?

A **Rule Manager** is a software system designed to define, manage, and execute rules in a structured and automated manner within a computational environment. Rules represent the decision logic that guides system operations, dictating behaviours and actions based on specific conditions. This tool is particularly useful in complex contexts where repetitive and standardized decisions need to be applied quickly and without errors.

Rules managed by a Rule Manager can range from simple logical conditions, such as “if... then...” statements, to complex models involving mathematical calculations, temporal analysis, or the combination of multiple criteria. A key characteristic of a Rule Manager is its ability to separate rule logic from the underlying application code. This approach enables both technical and non-technical teams to collaborate effectively. Domain experts, such as scientists in our case, can define rules using intuitive interfaces or declarative languages, while the system automatically executes them [Strawn, 2019].

In many cases, Rule Managers are integrated within a broader ecosystem that includes workflow management tools, data analysis platforms, and event monitoring systems. These technologies play a fundamental role in organizations that need to ensure compliance with complex regulations, optimize business processes, or respond quickly to dynamic events [Strawn, 2019].

1.2 How Does a Rule Manager Work?

A Rule Manager operates through a **Rule Engine**, which processes decisions based on predefined conditions applied to an incoming data set. The workflow typically begins with the definition of rules, which can be created using declarative languages or visual interfaces, allowing even users without advanced technical expertise to configure them. Rules can be straightforward, such as automatically approving a loan based on a customer's history and transaction value, or significantly more complex, incorporating temporal and geographical criteria. For example, a system may block a transaction if a credit card is used in two different countries within an hour [Duggan & Murray, 2018].

Once rules are established, the Rule Manager continuously gathers and monitors data from various sources, including institutional databases, real-time data streams such as online transactions or sensor readings, and externally generated events like API notifications or system logs. This step is essential to ensuring that rules are applied to relevant and up-to-date information. The ability to integrate multiple data sources makes the Rule Manager a key component in monitoring and automation systems, particularly in sectors like finance, cybersecurity, and Industry 4.0 [Strawn, 2019].

The core of the system is the rule processing stage, where the Rule Engine analyzes incoming data to determine which conditions are met. This process can be performed in batch mode, where large datasets are periodically evaluated, or in real-time, allowing the system to react instantly to new events and data updates. Advanced techniques such as conditional evaluation are applied to assess whether an action should be executed, while inference mechanisms allow the system to deduce new rules or actions based on existing ones and real-time information [Strawn, 2019]. These capabilities are particularly valuable in contexts requiring high levels of automation and rapid decision-making, such as fraud detection and industrial process optimization.

When a rule is satisfied, the Rule Manager executes the corresponding action, which may include sending notifications via email, SMS, or push messages, updating records in a database, activating workflows or processes in external systems, or generating reports for auditing and compliance purposes. A crucial aspect of this phase is the logging of each decision, ensuring that the decision-making process remains transparent and can be reviewed or audited later [Li & Li, 2018].

The adoption of Rule Managers has become essential across a wide range of applications, from fraud management and industrial automation to business intelligence and regulatory compliance. Their ability to operate in complex environments and make real-time decisions based on dynamic data makes them a strategic technology for enhancing operational efficiency and security in automated decision-making processes.

1.3 Applications and advantages of Rule Managers

Rule Managers are widely adopted across multiple sectors thanks to their ability to automate complex decisions, enforce compliance with predefined rules, and enhance the reliability of organizational processes. A particularly relevant application area is data management, where these tools ensure that information is efficiently validated, securely stored, and consistently optimized throughout its lifecycle—from acquisition to long-term preservation and sharing [Brock, 2018; Kallio et al., 2017; Zhou et al., 2019].

One of their primary functions is data validation, which involves automatic checks against specific quality and integrity criteria. For example, Rule Managers verify that data conforms to established formats, mandatory fields are correctly filled, and information remains consistent

across different sources, detecting anomalies or discrepancies when they occur [Chung et al., 2020]. Metadata management represents another critical use case: by automating the assignment of metadata according to standards such as Dublin Core or FDSN, Rule Managers guarantee that datasets are discoverable, interoperable, and reusable in compliance with international best practices [Wilkinson et al., 2016; Smith & Evans, 2018].

Beyond validation and metadata, Rule Managers play a key role in data classification, retention, and access control. They can automatically assign sensitivity levels, enforce GDPR requirements for personal data, and manage retention or backup policies by archiving obsolete information and securely deleting unnecessary records [Voigt & Von dem Bussche, 2017; Sato et al., 2018]. At the same time, they regulate permissions dynamically, restricting database access according to user roles, geographical origin, or authentication procedures [Hoffman & Sweeney, 2017]. They also support data transformations by aggregating, converting, or formatting information to ensure integration with heterogeneous systems, thereby facilitating advanced analysis and interoperability [Nguyen et al., 2019].

The adoption of Rule Managers provides substantial advantages beyond their direct applications. By automating repetitive decisions and processes, they save time, reduce human errors, and allow staff to focus on strategic activities [Nguyen et al., 2019; Liu et al., 2020]. Their configuration flexibility ensures that rules can be updated independently of the underlying code, making them particularly suitable for dynamic environments with frequently changing regulatory and operational requirements [Hassan & Leung, 2020]. Standardized rules significantly reduce error rates and enhance the reliability and consistency of business operations [Chung et al., 2020; Kallio et al., 2017].

Moreover, transparency is a built-in advantage: Rule Managers maintain detailed audit logs and decision records, enabling organizations to track operations and justify actions in compliance with data protection and accountability (frameworks such, for example, as GDPR) [Voigt & Von dem Bussche, 2017; Brock, 2018]. Their scalability is equally important, as they can manage large volumes of rules and transactions, easily adapting to the growth of businesses and the demands of global-scale operations [Zhou et al., 2019].

In summary, Rule Managers represent both a versatile application tool and a strategic advantage. They not only improve efficiency, reliability, and compliance but also provide organizations with the flexibility and scalability required to meet the challenges of modern digital ecosystems [Hoffman & Sweeney, 2017; Hassan & Leung, 2020].

1.4 Technologies and Tools

The implementation of a Rule Manager requires the use of advanced technologies and software tools capable of managing rules in a flexible, scalable manner while seamlessly integrating with existing infrastructures. These tools not only enable the efficient application of business policies and regulations but also offer advanced functionalities for process automation, continuous monitoring, and decision customization [Meyer & Högberg, 2021].

At the core of Rule Managers are Rule Engines, specialized software systems designed to process predefined rules based on configurable criteria [Hassan & Leung, 2020]. A Rule Engine allows decision logic to be separated from application code, making rules easily modifiable and adaptable to evolving business needs [Hoffman & Sweeney, 2017]. Furthermore, it supports structured formats for rule definition, such as YAML or specific languages like Drools, Jess and CLIPS, which enable logical conditions to be expressed declaratively [Brock, 2018]. Through API integration, Rule Managers can interact with other software systems, ensuring smooth and dynamic decision management [Nguyen et al., 2019].

In the context of data management, Rule Managers are closely connected with advanced

tools for data processing, management, and analysis. Among these, relational and NoSQL databases play a crucial role, as they allow for the rapid storage and retrieval of data required for rule evaluation. For instance, databases like MongoDB and PostgreSQL are often used to store input and output data from Rule Managers, ensuring high performance and reliability [Zhou et al., 2019]. Additionally, ETL (Extract, Transform, Load) tools such as Talend and Apache NiFi enable data extraction from heterogeneous sources, transformation according to predefined rules, and loading into target systems, ensuring a consistent and optimized data flow [Liu et al., 2020].

Another key technology is Data Lakes, infrastructures designed for storing vast amounts of raw data, which can be subjected to rules and policies in real time or on demand, providing flexibility and scalability even in high-workload environments [Chung et al., 2020].

Many modern Rule Managers are designed to operate in cloud environments, leveraging the power and elasticity of services like AWS Lambda, Google Cloud Functions, and Azure Logic Apps, which allow rule execution without the need to manage complex physical infrastructures. The adoption of microservices architectures provides an additional advantage, as it enables Rule Managers to be deployed as independent modular components, simplifying scalability, maintenance, and continuous updates [Hassan & Leung, 2020; Kallio et al., 2017].

Thanks to these technologies, organizations can implement Rule Managers that dynamically adapt to business needs while ensuring regulatory compliance and optimizing decision-making processes.

2. INGV Rule Manager: RUM

As seen, Rule Managers represent an essential solution for organizations that need to automate decisions and ensure compliance with specific regulations and desired standards. In short, they are tools designed to implement data policies on data stored in data centers while minimizing human intervention, making them ideal tools for the implementation of the INGV Data Management Plan [Fares et al., 2025].

Thanks to their flexibility, Rule Managers can be customized for different sectors and scenarios, improving efficiency and control. However, careful planning is fundamental when adopting these tools to manage complexity and ensure seamless integration with existing systems.

Currently available Rule Managers, whether open-source or commercial, often have a steep learning curve and require a complex installation process with important technical considerations. Additionally, they may involve significant investments that demand continuous sustainability, not to mention potential discrepancies in use cases compared to the specific needs of the scientific field.

For these reasons, the decision was made to develop an in-house Rule Manager-RUM—that is simple, lightweight, and easy to use and understand. The goal is to facilitate data management and maintenance within an open science data center in a production environment while maintaining the high-quality standards found in existing open-source and commercial Rule Managers.

2.1 RUM: Definition and General Overview

RUM was conceived as a lightweight yet powerful solution for managing data policies, designed to enforce predefined rules while maintaining operational efficiency and adaptability. Its architecture allows seamless integration into diverse environments without adding unnecessary complexity, making it particularly effective in contexts where data management

and security are top priorities. By automating policy enforcement, RUM ensures that data handling follows standardized and precise rules, reducing the risk of errors, inconsistencies, and non-compliance with internal policies or external regulations. Centralized monitoring further simplifies control processes, while integrated mechanisms for data integrity verification guarantee that information remains unaltered and reliable during storage and transfer phases. Beyond its role in policy enforcement, RUM offers advanced flexibility in executing predefined actions on data, such as files in mseed format. It enables the configuration of customized workflows that can be triggered by scheduled tasks—for instance, through crontab for periodic execution—or by specific events, such as the saving of a file in a dedicated directory. This event-driven and time-based adaptability allows RUM to fit a wide range of operational scenarios, processing only the data that is relevant according to user-defined criteria, including reference date, directory location, or other metadata.

By filtering and targeting specific datasets, RUM optimizes resource usage, avoiding unnecessary processing and significantly reducing execution times. This selective approach not only enhances performance but also ensures more efficient and transparent management of complex, evolving datasets. In essence, RUM combines automation, customization, and reliability to deliver a comprehensive solution for data governance, improving quality, traceability, and operational control.

2.2 Theoretical Foundation

The RUM system is based on a simple and modular model, emphasizing the flexibility and adaptability that characterize Rule Managers. Similar to Rule Engines, RUM organizes and manages complex decisions through the application of specific rules, but in a context tailored for data and file interaction.

At the core of RUM lies the concept of Policy (Data Policy), a framework that defines a general objective. In RUM's case, ensuring regulatory compliance, applied specifically to file monitoring and data management operations.

Each Policy in RUM consists of a series of Rules, which specify the conditions that must be met. For example, RUM's Rules might determine that a data-file should only be processed if it falls within a certain timeframe or is located in a designated directory.

When a Rule is satisfied, it triggers a set of Actions, which are practical operations executed on data. These Actions can range from file format validation (similar to data validation in data management systems) to copying, moving, or deleting files—a process akin to workflow automation in Rule Managers. The primary goal is to automate repetitive tasks, enhance operational efficiency, and reduce human errors, ensuring adherence to organizational policies and regulations.

A key aspect of RUM is the sequence in which Actions are applied, ensuring consistency and compliance. RUM guarantees that Actions are performed in a systematic, transparent, and traceable manner. This structured execution is particularly important in data governance, where tracking every action is essential for security and regulatory compliance [Fares et al., 2025].

Another important feature is RUM's modularity, allowing for easy implementation and future expansion. RUM enables the seamless introduction of new Rules and Actions, adapting to evolving business needs. This modular approach ensures agility in handling modifications without requiring a complete system overhaul—a crucial advantage in today's era of big data and digital transformation [Puglisi et al., 2018].

Policy, Rules and Actions: Rum model

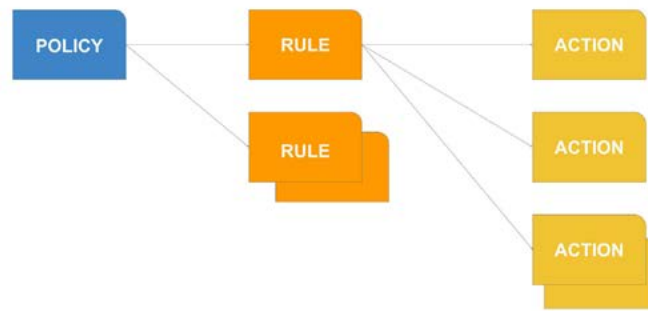


Figure 1 Policy, Rules and Actions: Rum model.

2.3 In practice

The system is designed to leverage a highly flexible and modular architecture, which facilitates the interaction and management of policies and rules. Policies and rules are defined using the YAML (Yet Another Markup Language) format, a configuration language that is extremely readable, even for non-technical people. This approach allows for the rapid definition of policies without the need for complex coding operations. For example, through YAML files, it is easy to describe the conditions that must be met, making the rule management process more transparent and accessible. The actions, which represent the concrete operations to be performed on data, are implemented as Python classes. These actions are invoked and applied based on the rules specified in the YAML files, establishing a direct connection between the rule logic and the operations on the data. This model resembles the functioning of a Rule Engine, where each rule defines the action to be executed when certain conditions are met.

The use of Python classes for actions ensures the possibility of implementing complex logic, similar to what is handled in rule engines, but with greater flexibility and customization. The system is designed to be easily extendable, allowing new actions to be added at any time without modifying the underlying architecture. It's enough to define new Python classes that implement the desired logic and insert them into the existing rules. This approach is similar to the ability of Rule Managers to integrate new rules without compromising the overall system structure, while ensuring a high degree of adaptability to different operational or business needs. The modularity of Python classes provides a dynamic environment in which it is possible to quickly respond to changes, implementing new features or optimizing existing ones.

An additional advantage of the proposed architecture is that the YAML configuration makes the management of policies and rules intuitive and understandable, even for team members without deep technical expertise. Similar to how a Rule Manager uses visual interfaces or declarative languages to simplify interaction with rules, the YAML-based system enables rapid and smooth adoption, even in business contexts where speed of implementation and ease of understanding are essential [Strawn, 2019].

2.4 Technologies Adopted

RUM is written in Python 3.10, a stable and widely used version of the Python programming language, known for its clear syntax, extensive library of modules, and ease of extension. Python is widely used in data management contexts due to its versatility and ability to integrate

with multiple systems and technologies. The use of Python 3.10 enables the leveraging of new features and performance improvements, making the system more robust and suitable for operations on large volumes of data. Additionally, the system utilizes container technology, specifically Docker, to simplify the installation process. The use of containers allows packaging the application and all its dependencies into an isolated, portable unit. This approach facilitates the system's installation in various environments, reducing issues related to version conflicts or incompatible configurations. Furthermore, containers allow for more efficient resource management of the system, which can easily be deployed on multiple servers or cloud computing environments [Wilkinson et al., 2016].

3. Definitions in RUM

3.1 Policy

Policies are sequences of simple operations (called Rules) that can be applied to data to manage or modify its content. Each policy is defined as a structured set of rules that, when executed in the predetermined order, lead to a specific outcome. Policies are not run concurrently, for this reason there isn't a mechanism that ensures consistency between policies. Policies can be adapted to meet different needs in a data management workflow, such as validation, transformation, or version management of the data.

The use of policies in RUM allows for the automation of complex processes, avoiding manual intervention and ensuring that all operations are performed consistently and repeatably. Policies are fundamental to applying best practices in data management, such as complying with regulations, ensuring quality, or applying filters to specific data.

In this Rule Management system, the definition of policies is done through the creation of configuration files in YAML format, where each policy is represented as a list of rules.

3.2 Rules

A Rule is a sequence of Actions that can be performed on data to modify its state or ensure that it meets certain criteria. Rules can range from simple operations, such as validating the format of a file, to more complex operations, such as applying compression algorithms, automatically renaming files, or storing them in specific storage systems.

Each rule is designed to be executed in a specific order, which can be determined by predefined conditions or specific events in the workflow. Rules are essential for process automation, as they define exactly what actions need to be taken and in which sequence to achieve a predefined goal. In a data management context, for example, a rule might specify that a data file, once collected, should first be validated to ensure it contains no errors, then transformed into a compatible format, and finally stored in a secure storage system.

3.3 Actions

An Action is a process applied to data without any knowledge of the current state of the workflow (that is actually managed by rules). In other words, each action must be able to be executed repeatedly on the same data without causing side effects or unwanted alterations to the results. Actions are the fundamental building blocks of RUM. Each action represents a specific operation, such as copying a file to another directory, compressing a file, assigning a

persistent identifier, or applying a validation algorithm. The workflow that implements a policy consists of a sequence of these actions, which are executed in order to ensure that the data is properly managed. Each action is designed to be simple and efficient.

Sometimes, to keep action scripts lightweight or for reusability reasons, it can be useful to divide the processes of an action into multiple steps, that is, multiple interrelated actions. In this case, you can use the Session object, which we'll see later, which allows the transfer of information between one action to another. In this rule management system, actions are implemented as functions or methods that can be dynamically invoked during the execution of a policy, configured through external configuration files (Figure 2). This approach allows new actions to be added to the system without modifying the base code, ensuring flexibility.

Policy, Rules and Actions: Rum implementation

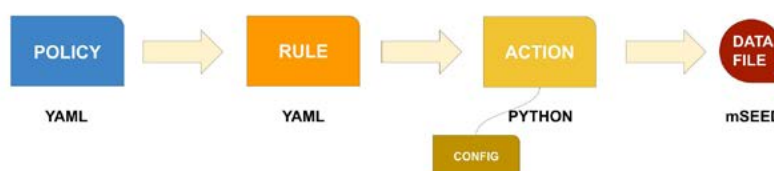


Figure 2 Policy, Rules and Actions: Rum implementation.

4. How RUM Works

The RUM project is structured into two main levels: the Rule-Engine, which represents the technological core of the system, and the application level (or project), which integrates and implements the defined business policies and rules. These two layers work in synergy to ensure automated and precise data processing, as discussed in the previous sections (Figure 3).

The Rule-Engine is the central component responsible for efficiently applying rules to incoming data. As mentioned earlier, the rule engine is designed to be highly configurable with the ability to quickly adapt to changes in policies or operational requirements without the need to rewrite code. Thanks to a series of modular components and configuration conventions, the Rule-Engine manages data operations, which are executed according to the order and logic established by the policies.

The application level, or project, focuses on the concrete implementation of policies, workflows, and actions associated with the rules. At this level, all components that interact with the Rule-Engine, manage data and ensure that operations are executed correctly as defined. Within the application level, the rule architecture, policy management, and action execution are all coordinated through software modules that guarantee process automation, as explored in the previous sections. Specifically, the ability to extend the system with new rules or actions, without affecting the underlying structure, is key to ensuring operational flexibility.

In practice, the application level interacts with the Rule-Engine to execute policies on the data. Similar to other advanced technologies for data management, such as database management systems or ETL engines, the application level ensures that all actions are applied consistently, and that the overall workflow is managed without errors. The Rule-Engine, therefore, provides the technological foundation to execute operations, while the application level manages its integration with external applications and users, making the system operational and functional. The division into two layers allows for a clearer and more modular management of the project,

where the Rule-Engine provides the technical structure, and the application level implements the specific behavior of the policies. This approach reflects the flexibility and modularity of the system, characteristics emphasized in the previous sections, which enable it to easily adapt to future needs or changes in requirements. The separation between the rule engine and the application level also facilitates system maintenance, as it allows one of the layers to be updated or modified without impacting the other.

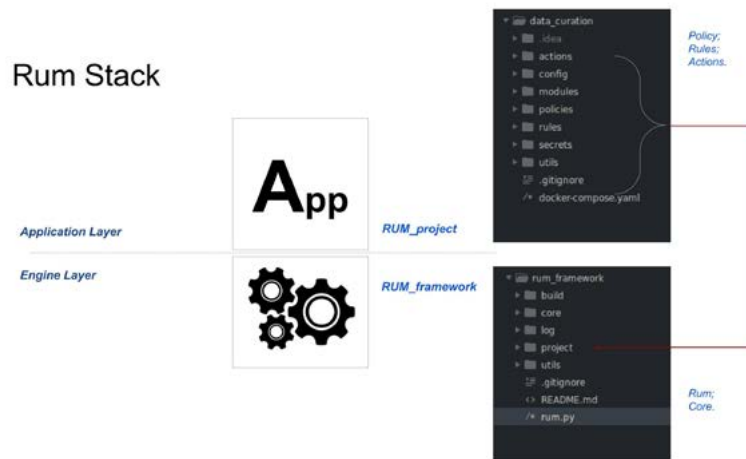


Figure 3 RUM Stack: Application and Engine layers.

4.1 Rule Engine Layer

The framework implementing the Rule-Engine is structured to ensure efficient processing of incoming data, combining various modular components that interact with each other to apply rules consistently. One of the distinctive features of this system is the inclusion of an ingestor, which acts as an input processor and is responsible for feeding the rule engine with the necessary data for processing. In analogy with what was discussed in the previous sections, where the fundamental role of the rule engine in the Rule Manager was highlighted, the ingestor acts as the first point of contact for the data, collecting it and preparing the ground for the application of the defined policies (Figure 4).

Once acquired, the data is transferred, which uses a configuration object to establish the behaviours to be applied to the incoming files. This step echoes the concept of flexibility and customization described earlier regarding policies and rules, where the system stands out for the ability to dynamically and intuitively configure workflows without the need for writing complex code. The ability to customize behaviours through explicitly defined configurations allows organizations to quickly adapt to changes in regulations or operational needs.

Once all system objects are initialized the data is passed to the sequencer, thanks to the dispatcher, a component that plays a role similar to that of “decision logic” in Rule Engine systems. The sequencer is responsible for executing the actions defined in the rules in order, ensuring that operations are applied in the correct sequence and that the overall workflow remains consistent. The use of the sequencer thus optimizes the automation process, reducing errors that could arise from disorganized execution or conflicts between rules. This approach mirrors the control logic and action execution discussed earlier, where automation not only improves efficiency but also ensures traceability and compliance with the established policies.

Rum engine layer

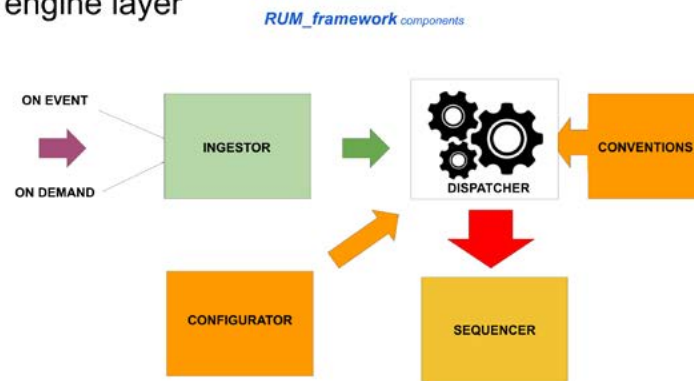


Figure 4 RUM engine layer.

4.2 Application Layer

As discussed earlier, the application layer in a RUM project is structured as a sequence of rules (established as a policy) that, in turn, are a sequence of actions applied to data. In practice, a policy defines a series of rules, and each of these rules consists of a sequence of actions that must be applied to the data. Each action, such as validating the file format, converting it, or applying processing algorithms, is designed to be an operation that does not depend on the previous state of the system or the complexity of the surrounding environment. This approach greatly simplifies the management of operations, as each action can be developed, tested, and modified separately without compromising the entire workflow.

Modularity and flexibility are key concepts that emerge in this application layer. Modularity allows for the creation of a set of reusable actions that can be easily combined in different ways depending on the needs of the workflow. For example, a file might initially require format validation, followed by conversion to another format, and finally stored in a specific path. These actions, although they may belong to different rules, are independent of each other and can be flexibly managed by the system.

This structure is similar to what happens in other data automation systems or workflow management systems, where each step (or action) is a single operation that can be combined with others based on business or operational logic. Additionally, as with rule engines discussed earlier, the application layer integrates seamlessly with the Rule-Engine (framework), which is responsible for ensuring the correct application of actions according to the defined policies. The sequence of actions is configured and controlled by the dispatcher, which applies the rules and manages the flow, as highlighted in the previous sections. This mechanism ensures that all operations are executed efficiently and in the correct order, automating data processing without requiring manual intervention.

In summary, the application layer of a RUM project is not just a set of actions to be performed on data, but a core component that enables the automated and modular application of defined policies. Its flexibility and modularity not only simplify the initial implementation but also allow for future expansions and modifications without affecting the overall system architecture. This level of abstraction enables the efficient and secure handling of even complex workflows, minimizing errors and optimizing data operations.

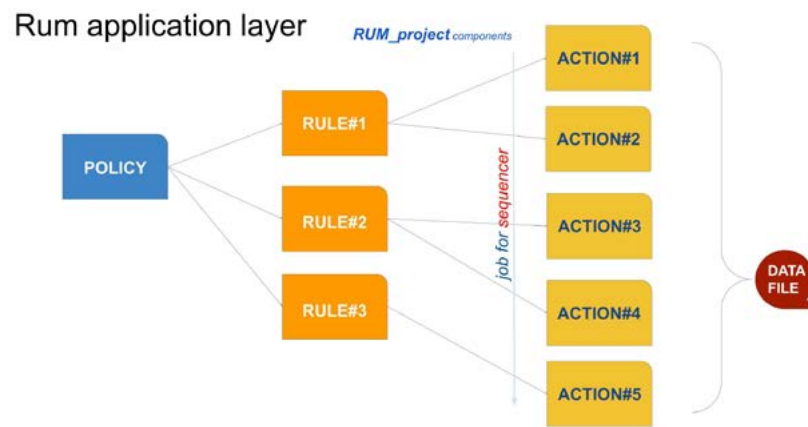


Figure 5 RUM application layer.

4.3 Global Objects of the RUM Framework

The RUM framework has some persistent objects that remain active throughout the entire lifecycle of the application. These objects are essential not only for configuration and monitoring but also for session management and operation logging (Figure 6). Unlike other solutions that might require separate management for each module, in the RUM framework, persistent objects centralize these tasks, thereby improving the overall system efficiency.

These global objects are fundamental because they enable unified management of system configuration, action traceability, and rule execution sessions. For example, as data is processed through actions defined in the application layer, global objects manage the workflow so that each step is properly monitored and recorded. This centralized process greatly simplifies the coordination of different activities, eliminating the need for complex interactions between modules or external systems.

Centralizing tracking and logging operations is another key feature that greatly enhances system transparency and management. Every operation, whether it's a data modification or a transformation, is logged, creating a traceable path that facilitates the verification of the results of the applied policies. This level of traceability not only ensures that all operations can be easily audited but also makes the system much more robust in terms of compliance and action monitoring.

When the system grows or new policies or rules are added, global objects allow for the integration and management of new elements without compromising the consistency or traceability of the system. This aspect of adaptability is fundamental to ensure that the RUM system can easily evolve in response to new operational needs or changes in data management policies.

In summary, the use of persistent and global objects within the RUM framework optimizes system management, centralizes monitoring and logging, and increases transparency and auditability. This approach creates a solid foundation for the automation of data processing and policy management.

RUM Framework GLOBAL OBJECTS

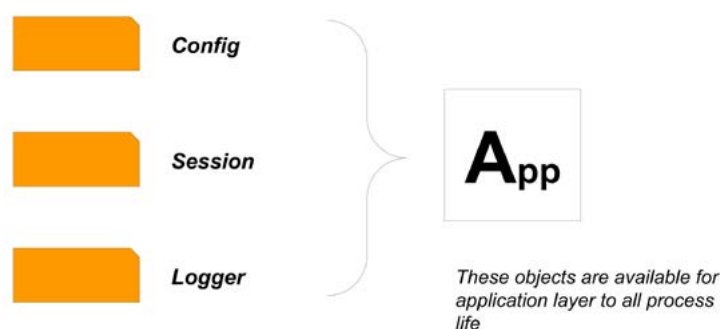


Figure 6 RUM Framework Global Objects.

5. RUM Workflow Configuration

5.1 Policy Configuration

Each YAML policy file represents a fundamental configuration for the system, as it defines how the rules and actions should be applied to the data. The file structure, located in the `/policies` directory, follows a precise schema that allows the system to efficiently and clearly read and interpret the policies.

The file name must adhere to a convention that helps maintain organization and clarity, so each policy file will have the format `policy-<policy-name>.yaml`. For example, a file related to data management policies might be named `policy-checkin.yaml`.

The content of this file is designed to be readable and easily editable, even by non-technical users, thanks to the use of the YAML format, which allows for a simple and structured representation of the configuration. Inside the file, the names of the rules involved in the policy and their execution sequence are declared. The basic configuration of a policy file could look like the following:

```
POLICY_NAME: checkin
POLICY_DESCRIPTION: do a CHECK-IN on data files to put them into
trusted archive
LOG_FILE: rum_checkin.log
LOG_LEVEL: INFO
WATCHED_ARCHIVE: "/var/lib/archive/incoming/"
RULES_SEQUENCE:
  '1': filechecks
  '2': extractmetadata
  '3': checkin2archive
```

Analysis of Elements

- **POLICY_NAME:** The name of the policy (in this case, `checkin`) uniquely identifies the set of rules and actions applied to the data. This allows for a quick understanding of the file's purpose, as seen in previous contexts where each rule or policy had a defined goal, such as "data validation" or "data transformation".
- **POLICY_DESCRIPTION:** A brief description clarifying the purpose of the policy. In this

example, the checkin policy is used to insert data files into a secure archive. This reflects the approach discussed earlier, where the policy defines a clear and tangible objective, such as “ensuring data quality” or “ensuring regulatory compliance”.

- **LOG_FILE:** Configuring a log file (here named `rum_checkin.log`) is essential for tracking all operations performed by the system. This is akin to the use of log objects discussed previously, which centralize the recording of operations to monitor system activity, facilitating the traceability of actions applied to the files.
- **LOG_LEVEL:** The log level (e.g., INFO) specifies the granularity of information recorded in the log file. Similarly to what was mentioned in rule management frameworks, it’s possible to configure the visibility of the workflow and execution details to optimize analysis and management of the applied policies.
- **WATCHED_ARCHIVE:** The directory `/var/lib/archive/incoming/` is the monitored path for incoming files. This parameter is particularly useful for directing data correctly within the policy management system, similar to the use of relational databases or data lakes for storing and managing data.
- **RULES_SEQUENCE:** The sequence of rules to apply to the data is defined by order numbers. In this case, the sequence is:
 - `filechecks`: the rule that performs initial checks on the files (e.g., format validation, metadata presence, etc.);
 - `extractmetadata`: the rule that extracts metadata from the file, ensuring that necessary information is correctly associated with the file itself;
 - `checkin2archive`: the rule that performs the “check-in” of the file into the designated archive.

This structure allows for the sequential application of rules, but as with rule engines, it’s possible to add or remove rules without altering the overall flow. The modularity and extendability of this configuration are reflected in the fact that the actions defined within the rules are independent and idempotent, allowing for smooth management of changes.

The architecture of the policy management system (as described in the RUM framework) remains consistent with the flexibility and scalability principles discussed in previous sections. Using YAML files to configure policies, the order and logic of operations are explicitly defined without needing to modify the system’s source code. Additionally, the sequence of rules and traceability through logs fit perfectly into a system aiming to reduce human error, increase automation, and ensure a secure and easily monitorable workflow.

Thus, the approach based on YAML configuration files not only optimizes policy management but also keeps the system open and easily adaptable to new requirements without compromising overall efficiency.

5.2 Rule Configuration

Each rule file will be configured in the `/rules` directory, following the format: `rule- <rule-name> .yaml`. These files are responsible for defining the actions to be performed during data processing. Each rule describes a sequence of actions to be applied to the files, along with a mapping between the rule tags and the action names, ensuring that the steps are connected and executed correctly.

Here’s an example of rule configuration:

```
RULE_NAME: filechecks
RULE_DESCRIPTION: performs some checks on the files before
```

```

proceeding with subsequent operations
ACTIONS_SEQUENCE:
    '1': PRE-FLY_CHECKS
    '2': SANITY_CHECKS
ACTION_MAP:
    PRE-FLY_CHECKS: preflychecks
    SANITY_CHECKS: sanitychecks
ACTION_RULE_CONFIG:
    preflychecks:
        IF_BAD_GOTO: some_action
    sanitychecks:
        IF_BAD_GOTO: some_action

```

Analysis of Elements

- **RULE_NAME.** The name of the rule (e.g., **FILECHECKS**) uniquely identifies the rule within the system. This name is fundamental for referencing the rule when used in a broader context, such as within a policy.
- **RULE_DESCRIPTION** (optional). The description provides a clear idea of what the rule does. In this case, **FILECHECKS** performs some checks on the files before proceeding with subsequent operations such as validation or metadata processing. This level of description helps quickly understand the role of each rule, as mentioned in the broader context of the RUM system, where every process element is well-defined and easily traceable.
- **ACTIONS_SEQUENCE.** This section specifies the order in which actions must be performed. Each number represents a phase in the sequence, and each phase is associated with a specific action to be performed. In the example:
 - 1 - **PRE-FLY_CHECKS**: the first action to be performed, related to preliminary checks.
 - 2 - **SANITY_CHECKS**: the second action, which verifies the file validity.

This sequence ensures that each operation is performed in the correct order, minimizing the risk of errors or inconsistent operations. The importance of a proper sequence of actions aligns with the concept of workflow consistency within the RUM system, where the dispatcher and sequencer are responsible for applying the rules in the defined order.

- **ACTION_MAP.** This section maps the tags to specific actions. Tags are keywords that identify each stage of the process, and they are used to refer to actions in more complex policies or for other organizational purposes. For example:
 - PRE-FLY_CHECKS** is mapped to the first action **preflychecks**.
 - SANITY_CHECKS** is mapped to the second action **sanitychecks**.

Tag mapping helps manage the workflow, especially when the system grows and policies become more complex. It is a strategy that enhances the system's maintainability since each tag can easily be reconnected to its respective action without confusion.

- **ACTION_RULE_CONFIG.** This section allows overriding configuration keys for actions so that the same action can be used for multiple rules with different setting values based on the rule being applied. In the example:
 - preflychecks**: the first action to be executed, related to preliminary checks.
 - IF_BAD_GOTO**: some action to perform if a condition fails.

The approach described in this rule configuration aligns with the principles of modularity and flexibility discussed earlier. As in the policies, where sequences of rules are used to determine the execution order, here too, actions are defined in a logical sequence that ensures correct data processing. The mapping of actions and their sequence is tightly coupled with the design of the RUM system, where the modular architecture allows rules to be easily adapted to the specific needs of each workflow or application. Additionally, the traceability of actions and operation monitoring is ensured by the rule file configuration, which keeps a record of the actions taken, as emphasized in the context of policies and logs. This system of rules and actions allows for clear and flexible definition and management of the necessary steps for data processing, ensuring consistency, scalability, and maintainability within the RUM.

5.3 Action Configuration

Each action must have a configuration file in the /config directory, and the name should follow this format: config- <action-name> .yaml. This file provides the relevant configuration values for the Python class associated with the action.

Example of Action Configuration:

```
ACTION_NAME: prefly_check
ACTION_DESCRIPTION: "PREFLY_CHECK performs a preliminary check on
the data before accepting it into the process:
  check if given file exist (source directory)
  check if it's auth network (by name of file)
  check if already in archive (target directory)"
ACTION_TYPE: validation
CONFIG:
TAG_FILE_ON: false
TAG_FILE: ".quarantine"
STATION_ENDPOINT: http://webservices.ingv.it/fdsnws/station/1/query?
STATION_ENDPOINT: "/var/lib/archive/good/"
ARCHIVE_BAD: "/var/lib/archive/bad/"
ARCHIVE_INCOMING: "/var/lib/archive/incoming/"
IF_BAD_GOTO: none
```

Analysis of Elements

- **ACTION_NAME.** The name of the action (e.g., `preflychecks`) uniquely identifies the action within the system, ensuring that it can be easily referenced within the rules. This name is directly linked to the Python class that implements the logic for executing the action, providing a clear connection between the configuration and the code performing the action.
- **ACTION_DESCRIPTION** (optional). The description provides a clear and concise idea of what the action does. In this example, `preflychecks` is responsible for verifying the files to ensure they meet certain minimum requirements before being processed. The description helps system administrators and developers quickly understand the purpose and function of the action without needing to delve into the code.
- **ACTION_TYPE** (optional). Indicates the type of operation the action performs. In this case, `validation` indicates that the action focuses on file validation, such as checking that the files are in the correct format and meet specific requirements (e.g., size and

extension). Using specific action types (e.g., validation, transformation, archiving) helps better structure operations and simplifies the management of actions during development and configuration.

- **CONFIG.** In this section, specific parameters for the action are defined, which may vary depending on requirements. Here's an analysis of the parameters included in the example:
 - **STATION_ENDPOINT.** The address of the service to be queried.
 - **ARCHIVE_BAD.** The path for files that are discarded from the process.
 - **IF_BAD_GOTO.** If present, indicates the next action to execute if a check fails.

These configurations (the contents of the **CONFIG** tag above) are available for each action through a “global configuration register”: `config['ACTION_CONFIG']['ACTION_NAME']`.

This approach centralizes configuration management and allows all actions to share a common configuration, simplifying workflow management and maintenance.

The action configuration follows an approach similar to that of rules and policies, where modularity and customization are key principles. Just as with policies and rules, the configuration of actions is completely separate from application logic, allowing system administrators to easily adapt the system's behavior to different needs without modifying the code. Flexibility and scalability are thus ensured.

Additionally, the use of YAML files for action configuration aligns with the idea of a system that is easily configurable and maintainable, as discussed in previous sections. The adoption of declarative configurations (like those in YAML files) makes the entire system more transparent and understandable, even for those who are not programming experts, promoting the integration of new workflows or updates to policies without rewriting the underlying code. An action performs checks and, depending on the result, can continue along different paths. The possible paths are defined at the rule level—for instance, using the IF-BAD-GOTO tag. This means that the action itself has no knowledge of the overall workflow or surrounding actions; it simply executes the configuration specified by the rule. It is important to note that rules override action-level configurations. As a best practice, all configuration keys should also be present in the action's configuration file. For example, if a rule sets IF_BAD_GOTO to “foo-action” while the action's configuration file has it set to “none,” the action will behave according to the rule when executed, adopting the “foo-action” path.

Each configuration file for actions plays a fundamental role in ensuring that the system is modular and easily adaptable to different needs. The specific configurations for each action clearly define the operations to be performed and allow for centralizing workflow management without compromising the transparency and traceability of operations, which are key aspects for the smooth functioning of the RUM system, as highlighted earlier.

6. Workflow Control in RUM

The concept of **SESSION** is crucial within the RUM framework, as it allows tracking and maintaining state throughout the entire cycle of applying a policy to a specific data. The session not only ensures the persistence of variables during the execution of rules and actions but also enables dynamic flow control, adapting to specific needs that may arise during data processing. This approach provides a higher level of flexibility, allowing the system to react in real-time to events or particular conditions without halting or reprocessing the entire process.

Actions behave according to a configuration defined by the rule. For example, when an action performs a check and the result is negative, the flow must jump to the corresponding process (e.g., error handling). In this case, the action reads the if-bad-goto variable defined in the rule

and reports it back to the sequencer (the policy/rule manager engine), which then makes the appropriate decision—namely, jumping to the specified action.

6.1 The Role of SESSION in Workflow Management

When a **Policy** is applied to a data, the **SESSION** is triggered and acts as a dynamic container for variables and states, which can be read or set by an action within the rule. This ensures a consistent context throughout the entire workflow.

Flow Control Features

In the context of the RUM framework, flow control logic can be implemented to determine how to proceed with the execution of actions. The main flow control operations are:

1. **Skip.** This operation allows skipping the execution of the very next action, maintaining the continuity of the workflow without executing the code related to that particular action. Flow control is managed by setting a session key that indicates a specific condition has been met, so the current action should not be executed.
2. **Example usage.** If a data does not meet certain requirements (e.g., format or size), the session might be configured to skip the file processing action, preventing unnecessary or erroneous processing, finally the sequencer implements this skip.
3. **Go to.** This command directs the execution to a specific action (forward only) in the workflow, skipping those between the current position and the target. It is useful when you want to skip intermediate operations based on certain criteria.
4. **Example usage.** If a specific validation action is successful, the system might jump to an “archiving” action, skipping any other “processing” actions, optimizing the process.
5. **Exit.** This operation allows terminating the policy execution early, canceling the entire execution cycle for that specific data-chunk. The “exit” operation is typically used when a serious error occurs or a condition makes further processing unnecessary.
6. **Example usage.** If a data-file fails a critical validation (e.g., the file is corrupted), the session can be configured to terminate immediately, preventing further processing that could compromise the quality of the workflow.

Integration with Rules and Actions

Flow controls like **skip**, **go to**, or **exit** are integrated at the action level but depend on the session configuration. When an action is executed, the system checks the session keys to determine if any flow control operations should be applied. Actions can then dynamically respond to variables such as the outcome of a validation, an error, or any other condition defined by the system.

6.1.1 Example of Flow Control in an Action

SKIP (skip)

When the value of the **SKIP** key is set to **1**, the workflow skips directly to the next action, continuing the execution cycle without executing the very next action.

GOTO (go to)

If the **GOTO** key is set to a specific action (the *name* of the action), the flow jumps directly to that action - via sequencer. It's important to note that the jump happens only forward, i.e., from a previous action to a subsequent one, according to the order specified by the policy and rule. It's not possible to jump to a previous action.

EXIT (exit)

If the **EXIT** key is set to **1**, the workflow terminates for the current data-chunk, the session is destroyed, and the process moves on to the next data-chunk. This is useful for halting the processing of a data-chunk when it is no longer necessary to continue.

This flow control management through **SESSION** and conditional checks strengthens the modularity and flexibility of the system, as highlighted earlier. Just like policies, rules, and actions, which define separate and autonomous workflows, the flow control within **SESSION** adds another layer of adaptability to the system. Each data entering the workflow can be treated in a personalized manner, responding dynamically to conditions and errors, without needing to stop or reprocess the entire system.

This ability to dynamically adjust to different conditions without compromising the integrity of the workflow is another example of how the RUM framework enables flexible, and reliable data management.

6.2 Use of Sessions

Sessions play a fundamental role in communication and managing information between different actions within a policy execution cycle. By allowing the persistent storage of variables throughout the entire process, sessions enable a more dynamic and interactive workflow between actions.

For example, an action might set a session variable with a value that will be later read and used by another action within the same policy cycle. This allows each action to access and react to the information produced by prior actions, creating a system of interdependence between operations that can be used to dynamically and precisely manage data processing.

Session variables can involve different types of information, such as the result of a format validation check on a file, the success of a conversion operation, or a value extracted during metadata analysis. A subsequent action might, for example, decide whether to execute further processing based on the value stored in a session variable. This approach makes the policy management system highly flexible, as actions can be conditioned by shared information that evolves throughout the process.

The communication between actions through sessions contributes to a more complex policy management, as it allows defining more intricate and reactive logics, where decisions are made based on state variables that change progressively. In fact, sessions provide a means to manage dependencies between actions, creating an operational sequence that can be precisely modeled according to workflow requirements.

This way, the system not only adapts dynamically to the specific needs of a work cycle but also automates and optimizes operations, minimizing manual intervention and reducing the potential for errors. Therefore, the use of sessions integrates perfectly with the overall vision of rule management, enabling advanced control over the flow of data and high efficiency in operations.

6.2.1 Important Note on the Use of the GOTO Key in Workflow Management

The **GOTO** key plays a pivotal role in managing the execution flow of actions within a policy. The value associated with this key represents a specific action to which the workflow can be directed. However, there's a fundamental rule: the value must always refer to an action that is "forward" in the sequence relative to the actions already defined in the policy and rule.

This principle ensures that while the flow can dynamically move toward a subsequent action (for example, from Action2 to Action5), jumping backward is not allowed: Jumping from Action5 to Action2 is not permitted. This approach prevents undesired cycles or regressions, maintaining the integrity and consistency of the workflow.

Controlling the flow using the **GOTO** key facilitates the creation of complex workflows, as it allows jumping between actions based on dynamic conditions or intermediate results. For instance, an action might determine that it is necessary to skip a subsequent operation and move directly to a more advanced stage of the process if certain criteria are met. This makes the system highly flexible and adaptable to complex scenarios, such as workflows dependent on continuously changing variables.

The "**forward-only**" rule for the GOTO key greatly enhances process automation. Dynamic conditions that occur during the execution of policies can trigger conditional branches, reducing the need for manual intervention and improving operational efficiency. Additionally, this approach ensures that each step in the workflow follows a logical sequence, preserving process transparency and reducing the likelihood of errors.

In summary, the use of the **GOTO** key in this context contributes to more effective and secure management of automated workflows, supporting the construction of data processing policies that are adaptable, dynamic and consistent with the objectives defined during the system design phase.

7. Installing the RUM Framework

7.1 Getting the Framework (Engine Level)

To obtain a copy of the RUM framework, you can directly download it from GitHub using the following link:

<https://github.com/INGV/rum-framework>

Alternatively, you can clone the repository by running the following command in your terminal:

```
git clone https://github.com/INGV/rum-framework.git
```

By cloning the repository, you'll get the latest stable version of the framework, which includes all the core functionalities required for managing policies and workflows. Cloning the repository is also useful to stay up to date with the developments and to have direct access to the source code.

7.2 Getting the Project (Application Layer)

Each project that you wish to use with the framework needs to be downloaded separately. To obtain a copy of a specific project, you can visit the following link on GitHub:

<https://github.com/INGV/rum-project/myproject>

Replace “myproject” with the name of the desired project.

Alternatively, you can clone the project repository using the following command:

```
git clone https://github.com/INGV/rum-project/myproject.git
```

Cloning the project repository will provide you with the necessary configuration files, policies, and rules required to apply the project-specific policy to your data. Each project is customized according to the data management needs and applicable policies.

7.3 Installation with Docker

For a quick installation of the RUM framework, you can use Docker to create an image of the application and run it in an isolated environment. This approach simplifies installation and execution without needing to manually configure the development environment.

7.3.1 Steps to build and run the Docker image

Clone the project repository (if you haven’t done so already):

```
git clone https://github.com/INGV/rum.git
cd rum
```

Build the Docker image from the project directory:

```
docker build . -t rum-process
```

This command creates the Docker image using the [Dockerfile](#) present in the repository. The `-t rum-process` option labels the image with the name `rum-process`, making it easy to identify later.

Run the Docker image. After the image is built, you can run the framework in a Docker container using the following command:

```
docker run -it rum-process
```

This will start the *RUM* framework inside an isolated environment, ready to be used without complicated configuration. Docker ensures that the application runs in a consistent environment, minimizing issues related to different operating system configurations.

7.3.2 Advantages of using Docker

Isolation. The application runs in a separate environment, reducing conflicts with other applications or system configurations.

Portability. You can run the framework on any operating system that supports Docker without worrying about configuration differences.

Easy updates. Docker simplifies the process of updating the application by allowing you to download or build a new image with the latest changes.

This solution makes the installation and execution of the framework faster and more reliable.

7.4 Expanded Installation

Installing the RUM framework with Docker is particularly useful in rapid development and testing contexts. Docker eliminates much of the complexity associated with manually configuring the environment, as everything needed to run the framework is contained within the Docker image. Additionally, using Docker allows you to easily manage different versions of the framework and test new changes without affecting the production environment.

If you prefer not to use Docker, you can run the framework directly in your local environment, ensuring that you have all the necessary dependencies installed, such as Python 3.10, support libraries (like requests or [PyYAML](#)), and any other packages required by the project.

7.4.1 Local Installation without Docker

Clone the repository:

```
git clone https://github.com/INGV/rum.git
```

```
cd rum
```

Set up a virtual environment (recommended to avoid conflicts with other Python projects):

```
python3 -m venv rum-env  
source rum-env/bin/activate # On Windows use `rum-env\Scripts\  
activate`
```

Install the required dependencies. You can install all the dependencies by running:

```
pip install -r requirements.txt
```

Verify the installation. Once the dependencies are installed, you can test the framework by running it locally with:

```
python3 run_framework.py --help
```

This approach is suitable if you need to customize the environment or prefer not to use Docker. However, it does require more manual setup and attention to dependencies, especially when working with different versions of libraries.

7.5 Next Steps

Once the installation is complete, you can proceed with configuring the framework and projects, defining policies and rules, and starting to process data according to the specified requirements. It's important to follow the documentation provided with each project to ensure optimal configuration and make sure that all dependencies are properly met.

Here are the main steps to get started.

1. Configure the Framework

- Edit the configuration files (config.yaml or any specific config file for your project) to adjust the framework's settings based on your environment and requirements.
- Set up the paths, system settings, and other configurations that will control the flow of data and the execution of policies.

2. Define Policies and Rules

- Define the policies and rules that will guide the processing of data. Policies should specify the conditions under which actions are executed, and rules should define how those actions are performed.
- Create or edit YAML files for policies and rules, making sure they align with the goals of your data processing tasks.

3. Configure Projects

- For each project, ensure that you have the correct configuration and file structures as specified by the project documentation.
- Customize the project settings to fit your specific needs, and ensure that all dependencies are installed (such as libraries or external tools).

4. Start Processing Data

- With the framework and projects configured, begin running the system to process data according to the defined policies and rules.
- Monitor the process and ensure that the workflow is proceeding as expected.

5. Test and Optimize

- Test the system using sample data to ensure that the policies and rules are working correctly.
- Optimize the framework by adjusting configurations or modifying the flow based on the results.

By following these steps, you'll be able to fully utilize the RUM framework, ensuring a smooth integration of your policies, actions, and projects for efficient data processing.

7.6 Command-Line Arguments Overview

When you launch the application, you can use various command-line arguments to further configure the behavior of the framework. Here is an overview of the available arguments:

7.6.1 Optional Arguments

-h, --help

Displays the help message and exits the application. By using this argument, the user can get an overview of all the other available arguments.

--mode MODE

Sets the execution mode of the framework. The available options are:

t: Terminal mode, where the application runs interactively in the terminal.

d: Daemon mode, where the framework runs in the background, constantly monitoring a folder for new file arrivals.

--files FILES [FILES ...]

Allows you to specify a list of specific files to be processed. You can list multiple files separated by spaces. For example:

```
--files file1 file2 file3
```

--archive ARCHIVE

Specifies the archive (e.g., an SDS archive or directory) containing the files to process. The archive can be provided as a full directory path where the mSEED files are located.

--julian-day JULIAN_DAY

Allows processing of files from a specific Julian day. For example, if you want to process files for the 230th day of the year, you would specify:

```
--julian-day 230
```

--policy-set POLICY_SET

Specifies a policy configuration file to be applied. A “policy-set” is a collection of rules defined for a process. For example, a policy file might contain a series of actions that are executed on the files being processed.

--version

Displays the current version of the program and exits.

These arguments provide flexibility in how the framework is executed and allow it to be tailored to different processing needs, whether you’re working in an interactive terminal session or running the system as a background service.

7.6.2 Usage Examples

Example 1

Process all files for day 230 in the directory 2015/IV and apply the “checkin” policy rules:

```
python3 rum.py --mode t --archive /mnt/archive/2015/IV --julian-day 230 --policy-set checkin
```

In this case, the command runs the framework in terminal mode, processing all files from the specified directory and applying the policy named “checkin.” The policy defines a set of rules for data archiving.

Example 2

Start a daemon that watches the specified directory in the configuration file. Each time a new file is added to the directory, the “checkin” policy is applied:

```
python3 rum.py --mode d --policy-set checkin
```

This command starts the framework in daemon mode, running in the background. Each time a new file is added to the monitored directory, the “checkin” policy is automatically applied to that file.

8. Final considerations

The RUM has proven to be a highly efficient and versatile solution for the automated management of data policies, particularly in the field of data management. Its simple and modular architecture offers numerous advantages, not only for optimizing data management operations but also for improving quality and ensuring compliance with both internal and external regulations. The ability to implement actions and rules in a flexible manner allows RUM to automate processes, reduce human error, and ensure adherence to organizational policies in a transparent and traceable way.

One of RUM's key strengths, however, lies in its modularity. This feature not only allows for easy system expansion with new rules and actions but also opens the door for implementations in various contexts beyond traditional data management. Thanks to its adaptable structure, RUM can be applied in a wide range of use cases, such as business process automation, decision management in complex scenarios, and even in workflow management within industrial settings. The system's modular design ensures that new functionalities can be integrated without disrupting the existing infrastructure, making it highly scalable for areas that are not strictly related to data management.

RUM's flexibility, combined with its ease of customization through the use of YAML files to define policies and rules, makes it particularly appealing for organizations that need a solution that can also be easily adapted to any context requiring rule-based decision automation and control. The potential of RUM as a decision management tool makes it an ideal candidate for integration into complex systems, such as monitoring and control systems, and for applications requiring high levels of customization and scalability.

In conclusion, RUM emerges as a versatile system that not only meets the demands of data management and compliance but also, due to its modularity, offers a robust and easily adaptable solution for a variety of complex scenarios. With the evolution of technologies and the expanding needs of businesses, RUM stands as a valuable resource for addressing the challenges of digitalization and decision automation.

Acknowledgments

The authors would like to express their sincere gratitude to Pasquale Cantiello and Pier Luigi Bragato for their careful review of this work and for their valuable comments and suggestions, which have greatly contributed to improving its clarity and overall quality.

Bibliography

- Brock J., (2018). *Data Governance in Practice: Policies and Processes for Managing Data*. Springer.
- Brock W.A., (2018). *Rule-based systems for decision automation*. *Journal of Software Engineering*, 45(3), 120-130.
- Chung C., Lee R., & Park S., (2020). *Data Validation Techniques for Large-Scale Systems*. Wiley.
- Chung T., Ho S., & Zhang L., (2020). *Data quality validation using rule engines*. *Data & Knowledge Engineering*, 127, 29-41.
- Duggan J., & Murray C., (2018). *Real-Time Decision Making: A New Era of Automated Rule Management*. *Journal of Intelligent Systems*, 24(3), 182-195.
- Duggan J., & Murray D., (2018). *Real-time Decision Making: Automated Systems and AI*. Springer.
- Fares M., Carluccio I., Danecsek P., Della Bina E., Franceschi D., Locati M., Mandiello A.G., Maniscalco M., Pintore S., Tavani F., (2025). *INGV EIDA ITALIAN NODE Data Management*

- Plan. Rapp. Tec. INGV, 492: 124, <https://doi.org/10.13127/rpt/492>
- Hassan M., & Leung J., (2020). Rule management for complex data systems. *International Journal of Data Science and Analytics*, 4(2), 75-88.
- Hassan M., & Leung K., (2020). *Digital Transformation and Data Management in the 21st Century*. Springer.
- Hennessy J.L., & Patterson D.A., (2019). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.
- Hoffman L., & Sweeney P., (2017). *Access Control and Data Protection Strategies*. Oxford University Press.
- Hoffman R., & Sweeney T., (2017). Access control models and their implementation in data management. *Information Systems Security*, 22(5), 205-217.
- Kallio J., Ahonen J., & Tikkanen K., (2017). *Data Management Strategies in the Digital Age*. Elsevier.
- Kallio J., Pulkkinen A., & Aho T., (2017). Implementing flexible rule-based systems in business environments. *International Journal of Business Informatics*, 15(4), 54-68.
- Li S., & Li X., (2018). Rule-Based Expert Systems: A Review. *Expert Systems with Applications*, 95(3), 121-131.
- Liu Y., Chen X., & Wong P., (2020). ETL Processes and Data Integration Strategies for Big Data. *ACM Transactions on Information Systems*.
- Liu Y., Zhou Z., & Xu L., (2020). Automation and flexibility in rule-based decision making. *Journal of Enterprise Information Systems*, 16(2), 112-127.
- Meyer C., & Högberg M., (2021). *Automated Rule Management in Enterprise Systems*. Springer.
- Meyer M., & Högberg S., (2021). Technological advancements in rule-based systems for automated decision-making. *Journal of Software and Systems Engineering*, 37(4), 204-215.
- Nguyen T., Li X., & Zhou H., (2019). *Data Transformation for Business Intelligence Applications*. Elsevier.
- Nguyen T., Tran D., & Pham L., (2019). Automating data transformation processes using rule-based systems. *International Journal of Database Management Systems*, 11(2), 25-38.
- Puglisi G., et al., (2018). *INGV Data Policy and Management Guidelines*. Istituto Nazionale di Geofisica e Vulcanologia; <https://hdl.handle.net/2122/14886>
- Sato H., Lee C., & Iida S., (2018). Data archiving and backup policy enforcement using rule engines. *Journal of Data Storage*, 16(4), 77-92.
- Sato Y., Tanaka K., & Nakamura H., (2018). *Data Retention and Backup Policies in Enterprise Systems*. ACM Transactions on Information Systems.
- Smith A., & Evans P., (2018). *Metadata Management for Business and Technology*. CRC Press.
- Smith J., & Evans G., (2018). Metadata management in rule-based data systems. *Journal of Information Management*, 32(7), 45-56.
- Strawn G., (2019). *The Future of Computing Performance: Game Over or Next Level?* National Academies Press.
- Taylor J., et al., (2020). Global collaboration in scientific data management systems. *Data Science International*, 17(2), 78-93.
- Voigt P., & Von dem Bussche A., (2017). *The General Data Protection Regulation (GDPR): A guide for data protection professionals*. Springer.
- Wilkinson M.D., Dumontier M., Aalbersberg I.J., et al., (2016). The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, 3, 160018.
- Zhou L., Wang H., & Zhang Y., (2019). *Big Data and Compliance: Methods and Applications*. IEEE Press.
- Zhou Z., Liu Y., & Xu L., (2019). Rule-based data validation in modern enterprise systems. *International Journal of Enterprise Information Systems*, 15(4), 31-45.

QUADERNI di GEOFISICA

ISSN 1590-2595

<https://istituto.ingv.it/it/le-collane-editoriali-ingv/quaderni-di-geofisica.html/>

I QUADERNI DI GEOFISICA (QUAD. GEOFIS.) accolgono lavori, sia in italiano che in inglese, che diano particolare risalto alla pubblicazione di dati, misure, osservazioni e loro elaborazioni anche preliminari che necessitano di rapida diffusione nella comunità scientifica nazionale ed internazionale. Per questo scopo la pubblicazione on-line è particolarmente utile e fornisce accesso immediato a tutti i possibili utenti. Un Editorial Board multidisciplinare ed un accurato processo di peer-review garantiscono i requisiti di qualità per la pubblicazione dei contributi. I QUADERNI DI GEOFISICA sono presenti in "Emerging Sources Citation Index" di Clarivate Analytics, e in "Open Access Journals" di Scopus.

QUADERNI DI GEOFISICA (QUAD. GEOFIS.) welcome contributions, in Italian and/or in English, with special emphasis on preliminary elaborations of data, measures, and observations that need rapid and widespread diffusion in the scientific community. The on-line publication is particularly useful for this purpose, and a multidisciplinary Editorial Board with an accurate peer-review process provides the quality standard for the publication of the manuscripts. QUADERNI DI GEOFISICA are present in "Emerging Sources Citation Index" of Clarivate Analytics, and in "Open Access Journals" of Scopus.

RAPPORTI TECNICI INGV

ISSN 2039-7941

<https://istituto.ingv.it/it/le-collane-editoriali-ingv/rapporti-tecnici-ingv.html/>

I RAPPORTI TECNICI INGV (RAPP. TEC. INGV) pubblicano contributi, sia in italiano che in inglese, di tipo tecnologico come manuali, software, applicazioni ed innovazioni di strumentazioni, tecniche di raccolta dati di rilevante interesse tecnico-scientifico. I RAPPORTI TECNICI INGV sono pubblicati esclusivamente on-line per garantire agli autori rapidità di diffusione e agli utenti accesso immediato ai dati pubblicati. Un Editorial Board multidisciplinare ed un accurato processo di peer-review garantiscono i requisiti di qualità per la pubblicazione dei contributi. RAPPORTI TECNICI INGV (RAPP. TEC. INGV) publish technological contributions (in Italian and/or in English) such as manuals, software, applications and implementations of instruments, and techniques of data collection. RAPPORTI TECNICI INGV are published online to guarantee celerity of diffusion and a prompt access to published data. A multidisciplinary Editorial Board and an accurate peer-review process provide the quality standard for the publication of the contributions.

MISCELLANEA INGV

ISSN 2039-6651

<https://istituto.ingv.it/it/le-collane-editoriali-ingv/miscellanea-ingv.html>

MISCELLANEA INGV (MISC. INGV) favorisce la pubblicazione di contributi scientifici riguardanti le attività svolte dall'INGV. In particolare, MISCELLANEA INGV raccoglie reports di progetti scientifici, proceedings di convegni, manuali, monografie di rilevante interesse, raccolte di articoli, ecc. La pubblicazione è esclusivamente on-line, completamente gratuita e garantisce tempi rapidi e grande diffusione sul web. L'Editorial Board INGV, grazie al suo carattere multidisciplinare, assicura i requisiti di qualità per la pubblicazione dei contributi sottomessi. MISCELLANEA INGV (MISC. INGV) favours the publication of scientific contributions regarding the main activities carried out at INGV. In particular, MISCELLANEA INGV gathers reports of scientific projects, proceedings of meetings, manuals, relevant monographs, collections of articles etc. The journal is published online to guarantee celerity of diffusion on the internet. A multidisciplinary Editorial Board and an accurate peer-review process provide the quality standard for the publication of the contributions.

Progetto grafico
Barbara ANGIONI
Istituto Nazionale di Geofisica e Vulcanologia

Ufficio Editoriale
Francesca DI STEFANO | Coordinatore

Segreteria di redazione
segreteria.collane-editoriali@ingv.it

Editing | Produzione | Grafica redazionale
Barbara ANGIONI
Massimiliano CASCONI
Francesca DI STEFANO
Patrizia PANTANI

© 2026 the Author(s). All rights reserved. Open Access



Creative Commons Attribution 4.0 International



ISTITUTO NAZIONALE DI GEOFISICA E VULCANOLOGIA