INFORE Interactive Extreme-Scale Analytics and Forecasting

Refined Architecture, Initial System Prototype and Design of Software Stacks
Work Package 4 Tasks 4.2-4.4 Deliverable D4.2

Authors

Ralf Klinkenberg, David Arnu, Edwin Yaqub, Fabian Temme, Mate Torok
RapidMiner GmbH

Antonios Deligiannakis, Alkis Simitsis, Nikos Giatrakos
Athena Research & Innovation Center

Elias Alevizos, Nikos Katzouris
National Center for Scientific Research “Demokritos”

European Commission Horizon 2020 European Union funding for Research & Innovation	Project supported by the European Commission Contract no. 825070	WP4 T4.2-4.4 Deliverable D4.2	**Doc.nr.:**	WP4 D4.2
			Rev.:	1.0
			Date:	30/04/2020
			Class.:	Public

Distribution list:

Group:	Others:
WP Leader: RapidMiner Task Leader: RapidMiner	Internal Reviewer Partner: Istituto Italiano di Tecnologia (IIT), MarineTraffic (MT) INFORE Management Team INFORE Project Officer

Document history:

Revision	Date	Section	Page	Modification
0.1	26/03/2020	1-4	1-5	Creation
0.2	29/03/2020	3-4	1-20	Architecture Design, Software Specification
0.3	03/04/2020	3-4	8-29	Optimizer and Synopses Data Engine incorporation
0.4	07/04/2020	1-4	All	Self-Review
0.5	09/04/2020	All	All	Submitted for internal review
0.7	15/04/2020	8	38-60	Appendix added
0.8	20/04/2020	-	-	Internal review comments received
0.9	28/04/2020	All	All	Internal review comments incorporated
1.0	29/04/2020	All	All	Final modifications by the Coordinator

Approvals:

First Author:	Ralf Klinkenberg (RM)	Date:	07/04/2020
Internal Reviewer:	Gian G. Tartagia (IIT), Marios Vodas (MT)	Date:	20/04/2020
Coordinator:	Antonios Deligiannakis (Athena)	Date:	29/04/2020

Table of contents:

1 Executive Summary

This deliverable describes the prototype implementation of the INFORE Architecture, which is defined in the deliverable D4.1 (Month 12) of the INFORE project. The following sections detail how the main objectives for the INFORE Architecture are achieved by initial implementations and prototypes of the components. The INFORE Architecture aims for a holistic, pluggable, extensible framework which supports the following objectives:

i. supporting the non-programmer data analyst in rapid setup of streaming workflows tailored for her application scenario needs by providing graphical workflow design facilities,
ii. automating the tuning of the underlying Big Data platform infrastructure that materializes the visually designed workflow as well as the provisioned physical resources in a way that optimizes specific performance metrics,
iii. providing real-time, interactive machine learning and data mining tools that can be leveraged by the designed workflows,
iv. enhanced interactivity via data summarization and approximate query processing techniques,
v. distributed complex event processing and forecasting techniques to not only detect business events of interest as soon as they occur, but also forecast their occurrence well in advance.

These goals are achieved by implementing the following components:

- Graphical Editor Component
- Connection Component
- Manager Component
- Optimizer Component
- Synopsis Data Engine Component
- Interactive Online Machine Learning Component
- Complex Event Forecasting Component

The functionality described in deliverable D4.1 is achieved by the respective implementations of the components and the interactions between them.

The RapidMiner Studio software is extended by the streaming extension. The extension provides the implemented Graphical Editor Component, the Connection Component and the Manager Component.

In addition, it provides the integration with the Optimizer Component, the Synopsis Data Engine Component and the Interactive Online Machine Learning Component. The implementation of the integration of the Complex Event Forecasting Component is ongoing and will be reported in Deliverable D4.3 (Month 32).

Due to these implementations, the streaming extension enables users of the INFORE Architecture to design streaming analysis workflows via logical streaming operations without the need for programming or scripting skills. The designed workflow abstracts away the streaming logic from actual, physical implementations; there is no need for the user to concern himself with such details. The streaming extension of RapidMiner provides the *Streaming Optimization* operator which utilizes the capabilities of the Optimizer Component to perform an optimization of the designed workflow. The resulting physical workflow is then visualized in RapidMiner Studio and can be dispatched for execution by using the new *Streaming Nest* operator. This operator implements the functionality of the Manager Component by being able to create streaming jobs specific to streaming platforms and submitting those to the respective clusters.

The streaming extension also provides the new operators *Synopsis Data Engine*, and *Online Machine Learning*. They make use of the interfaces, which are described in the deliverable D4.1, to integrate functionality provided by above-mentioned components in the streaming analysis workflow. The components can be configured through the graphical user interface of RapidMiner Studio and be leveraged in the designed workflow.

This deliverable focuses on the integration of the various components that together constitute the initial prototype of the entire INFORE Architecture. The implementations and inner workings are detailed for the Optimizer Component in deliverable D5.1 (Month 16, together with this deliverable), the Synopsis Data Engine Component in deliverables D6.1 (Month 12) and D6.3 (Month 24), the Interactive Online Machine Learning Component and the Complex Event Forecasting Component in deliverables D6.2 (Month 16, together with this deliverable), D6.4 and D6.5.

2 Introduction

INFORE’s project objectives target the processing of large-scale data that is increasingly being produced as streams by a variety of industrial and scientific applications. To handle these massive data streams, transforming them, training and applying analytics functions on them, a flexible, extensible and customizable approach is needed. This approach should allow end users to easily model and manage streaming analysis workflows. Considering the evolution of streaming technologies, the subtle differences in their capabilities, as well as the complexity of streaming use cases, INFORE envisions a cross-platform framework approach to visually create streaming processes that can integrate complex disparate components or services using a flexible integration approach driven by a common middleware to act as an interfacing backbone. INFORE architecture incorporates a generic, technology-agnostic and platform-abstract representation of visual workflows as streaming graphs, which are transformed and translated to multiple supported streaming platforms. This highly extensible and pluggable approach makes INFORE streaming workflows future-proof in terms of fast changing technology landscape. The INFORE architecture has the means to optimize processes at different levels, including reconfiguring process parameters, splitting up processes for placement on potentially multiple streaming platforms - given real-time usage statistics on infrastructural capacity of available platforms, functional (quality) objectives and non-functional requirements of streaming processes. The INFORE functional prototype deploys these optimized processes in a seamless fashion, which enables analysis of results in an interactive and continuous manner.

A major challenge in INFORE is to bring the various components of the architecture together into one system, that has a common design principle and follows a unified user experience. The functional prototype introduced in the following sections, is the result of this effort to develop a consolidated system based on the envisioned architecture of the project team. It integrates the user-friendly design of streaming processes and allows the usage of different streaming technologies, without requiring an in-depth knowledge of the underlying technologies. The enhancements rendered by the Optimizer Component allow us to adapt and adjust the processes to different workflow layouts. This is arguably a highly novel approach since it bridges the gap between design-time and execution-time by providing the tooling to optimally harness available physical infrastructure. In addition, the integration of the components for the synopsis data engines and online machine learning using a common interfacing middleware allowed us to integrate disparate asynchronous components. In short, INFORE has leveraged state of the art technologies but more importantly, INFORE offers innovative functionalities that are not available in contemporary systems. Thus it can be claimed that INFORE has indeed extended the state of art to various degrees.

The following sections will describe the different parts of the architecture, the technical and engineering aspects of various components and how they are implemented in the current version of the prototype. While still in an early stage, the system already clearly shows the flexibility and the expressiveness of the INFORE framework.

3 Integration of the Components of the INFORE Architecture

This section describes how the developed prototype implements the structure and concepts of the INFORE Architecture. This structure and concepts were defined in the deliverable D4.1 (Month 12). As the definition of the Architecture is the theoretical background of the initial prototype the Sections 4 and 5 of the deliverable D4.1 are added to the appendix of this document (see Sections 8.1 and 8.2) for convenience. The main concept of the INFORE Architecture is also summarized in Section 3.1. This section also includes a refinement of the Architecture, needed by the experience gained while developing the prototype.

Section 3.2 focuses on how the different components are integrated, how they interact with each other and how this is used to build the first functional prototype of the INFORE Architecture. Implementation details on the specific components and their integrations are described in Section 4. Section 3.3 discusses how far the functionality of the INFORE Architecture is achieved by the initial prototype.

As mentioned, the streaming extension was added to the RapidMiner Studio and implements the Connection Component, the Graphical Editor Component and the Manager Component. The streaming extension also implements integrations of the Optimizer Component, the Synopsis Data Engine Component and the Online Machine Learning Component. These integrations enable the interaction with the specific components in the INFORE Architecture. Hence, the streaming extension is the implementation of the prototype of the INFORE Architecture, which is described in this document.

3.1 INFORE Architecture Design

Figure 1 shows an overview of the concept of the INFORE Architecture, designed in the deliverable D4.1. This subsection will give a short recap of the architecture and introduce an update to the concept. The user interface of the INFORE Architecture comprises the Graphical Editor, which provides the *Streaming Nest* operator, standard streaming transformation operators and the two “Component Operators” (*Synopsis Data Engine, Online Machine Learning*). It also provides various connection objects that store necessary information for interacting with specific clusters. Those clusters are either responsible for data streaming or the execution of streaming jobs. Users can utilize these operators and connection objects to design streaming analysis workflows in a visual environment.

The designed workflow is then sent to the Manager Component. The Manager instructs the Optimizer Component to optimize the workflow. To do this, it retrieves information about available resources in the computing clusters. The optimized plan gets then executed on one or more computing cluster(s). After deployment, jobs can be monitored by the Manager Component.

As an update on this concept, the *Streaming Optimization* operator is introduced in this deliverable. It represents the optimization of the streaming workflow by the Optimizer Component, in the Graphical Editor. The user still designs the streaming workflow in the same manner, but now in the subprocess of the *Streaming Optimization* operator. All connection objects are provided to streaming execution backends, which can potentially be used for job execution. The Optimizer Component performs the optimization of the workflow and decides on the placement of the logical operators on the computing clusters. The *Streaming Optimization* operator creates *Streaming Nest* operators and links those with corresponding connection objects. This defines the computing cluster and big data technology on which the corresponding operators are to be executed. For a more detailed description of these operators see the following subsections.

This deliverable also concretizes how logical streaming operators are implemented and used in the INFORE Architecture. The streaming operators (see Section 4.2 for a list of the operators) are implemented as logical streaming operations. They must be placed inside a *Streaming Nest* operator (or inside a *Streaming Optimization* operator, which by itself places them again in *Streaming Nest* operators). The connection object provided to the *Streaming Nest* operator defines the used streaming technology (the actual physical implementation). This causes an update of the JSON interface used in the communication between Manager Component and Optimizer Component (see Section 4.4).

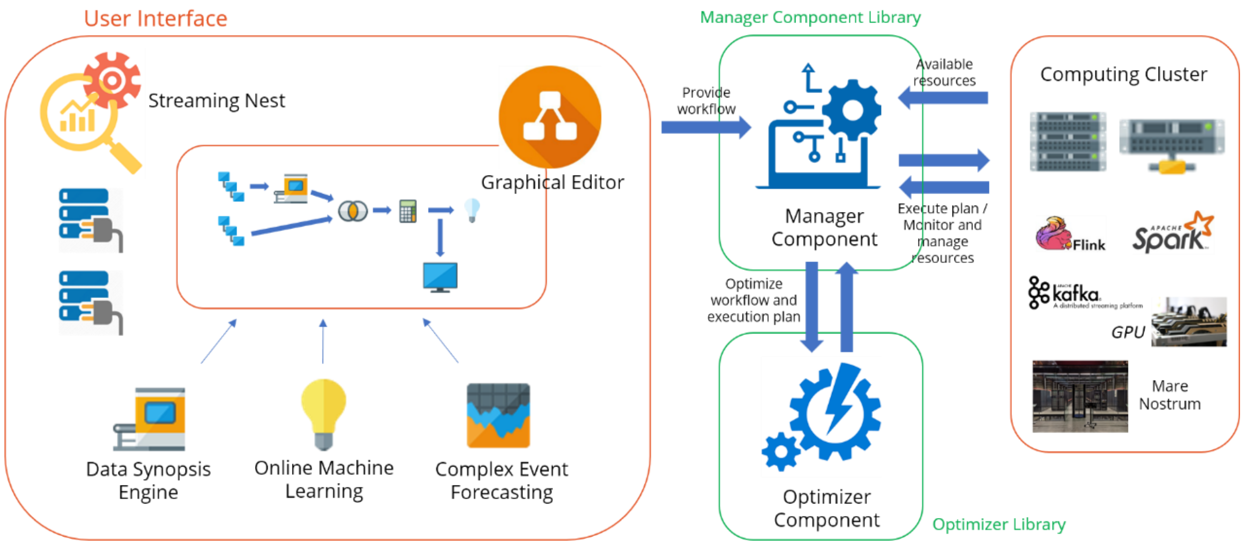


Figure 1: Overview of the conceptual design of the INFORE Architecture (image from deliverable D4.1).

3.2 Integrations of the Components of the INFORE Architecture

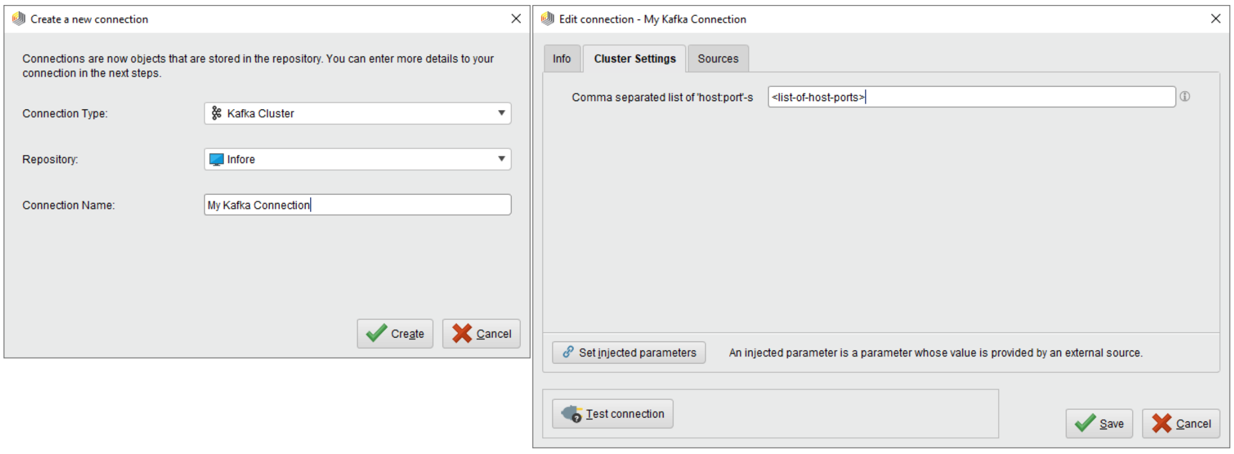
This section describes the integrations of the different components in the prototype of the INFORE Architecture.

3.2.1 Connection Component Integration

To integrate the Connection Component new connection object types are added to the streaming extension of RapidMiner Studio. Connection objects (from now on called connections) in RapidMiner Studio store all necessary information to connect to a specific type of service, typically over network. They can be stored in the same way as other objects in the RapidMiner Studio repository and they are handled in the process panel in the same way. They can be retrieved from the repository with the *Retrieve* operator, be connected to different operators or placed into subprocesses by utilizing corresponding ports. The *Multiply* operator is suitable for duplicating connection information in the current process, so that the same connection can be used at different operators. They can also be shared with others, for example by using RapidMiner Server. Information in connections can be encrypted and user specific entries (e.g. username and password) can be left out from the connection object and injected by the corresponding user. For more details about connection objects, see the documentation of connection objects in the official RapidMiner Studio documentation[1].

The streaming extension adds three new connection types to RapidMiner Studio: *Kafka Connection, Flink Connection* and *Spark Connection*. They respectively allow configuring communication channels to Kafka, Flink and Spark clusters. *Kafka Connections* are used by the new operators *Kafka Source* and *Kafka Sink* to bring input data streams into the workflow and output processed data as a stream. They also play a central role in the new *Synopsis Data Engine* and *Online Machine Learning* operators to facilitate interaction with the corresponding components. Figure 2 shows the graphical interface to create a *Kafka Connection* object. As shown, the graphical interface empowers users with the ability to provide the necessary information to connect to the Kafka cluster.

[1] https://docs.rapidminer.com/latest/studio/connect/


Create a new connection

Connections are now objects that are stored in the repository. You can enter more details to your connection in the next steps.

Connection Type: Kafka Cluster

Repository: Infore

Connection Name: My Kafka Connection

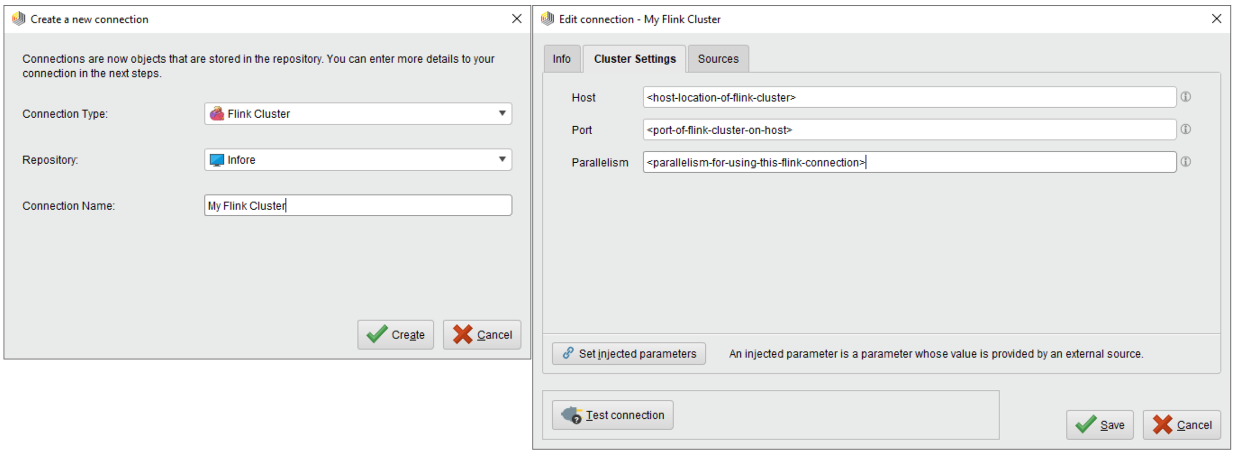
Edit connection - My Kafka Connection

Comma separated list of 'host:port'-s <list-of-host-ports>

An injected parameter is a parameter whose value is provided by an external source.

Figure 2: Graphical interface to create a new *Kafka Connection*. The type, the location (repository) and the name of the connection must be specified, as well as the Kafka specific settings.

Flink Connections are used by the new *Streaming Nest* operator to deploy a streaming analysis workflow on a Flink cluster. Using a *Flink Connection* for the input of the *Streaming Nest* operator therefore defines the used technology of the logical streaming operators in the *Streaming Nest* operator. Figure 3 shows the graphical interface to create a *Flink Connection* object. The necessary information to connect to the Flink cluster can easily be provided.


Create a new connection

Connections are now objects that are stored in the repository. You can enter more details to your connection in the next steps.

Connection Type: Flink Cluster

Repository: Infore

Connection Name: My Flink Cluster

Edit connection - My Flink Cluster

Host <host-location-of-flink-cluster>

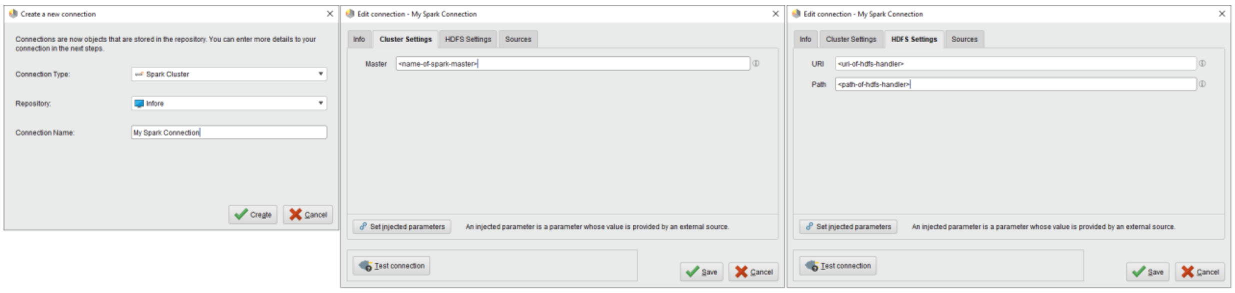
Port <port-of-flink-cluster-on-host>

Parallelism <parallelism-for-using-this-flink-connection>

An injected parameter is a parameter whose value is provided by an external source.

Figure 3: Graphical interface to create a new *Flink Connection*. The type, the location (repository) and the name of the connection must be specified, as well as the Flink specific settings.

Like *Flink Connections*, *Spark Connections* are used by the new *Streaming Nest* operator to deploy a streaming analysis workflow on a Spark cluster. Hence, they also define the used technology of the logical streaming operators in the *Streaming Nest* operator. Figure 4 shows the graphical interface to create a *Spark connection* object. Like for Kafka and Flink, the necessary information can be easily provided as well. For more details about the deployment of streaming jobs by the *Streaming Nest* operator see Section 3.2.3 and 4.3. For the implementation details of the connection objects see Section 4.1.



Connections are now objects that are stored in the repository. You can enter more details to your connection in the next steps.

Connection Type: Spark Cluster

Repository: Infore

Connection Name: My Spark Connection

Master <name-of-spark-master>

An injected parameter is a parameter whose value is provided by an external source.

URI <uri-of-hdfs-handler>

Path <path-of-hdfs-handler>

An injected parameter is a parameter whose value is provided by an external source.

Figure 4: Graphical interface to create a new *Spark Connection*. The type, the location (repository) and the name of the connection must be specified, as well as the spark specific settings.

3.2.2 Graphical Editor Component Integration

To integrate the Graphical Editor Component, the streaming extension of RapidMiner Studio is developed. It allows to utilize the already existing graphical editor of RapidMiner Studio to design streaming analysis workflows in a simple to use, visual design approach. The streaming extension adds several new operators to RapidMiner Studio. To define a streaming analysis workflow, the new *Streaming Nest* operator can be used. It is a subprocess operator, which means that it has an internal subprocess where operators can be placed. All streaming operators of a streaming analysis workflow are placed inside the *Streaming Nest* operator.

Besides the *Streaming Nest*, the new extension adds several stream operators, which provide some standard streaming analysis functionality. The operators *Kafka Source* and *Kafka Sink* enable reading and writing of Kafka topics in a streamed way. The operators *Duplicate Stream*, *Join Streams* and *Connect Streams* enable stream flow handling in a streaming analysis. The operators *Aggregate Stream*, *Filter Stream* and *Map Stream* enable standard streaming transformations.

All operators are designed as logical operators. They define only the logic of the streaming analysis, not the actual technology used to achieve this. This is done by providing the corresponding streaming execution connection to the *Streaming Nest* operator (see Section 3.2.1). The *Streaming Nest* send the designed workflow, together with the connection object to the Manager Component, which creates a backend specific streaming job from the logical streaming workflow. This backend specific streaming job is then deployed on the corresponding cluster. For more details about this, see Section 3.2.3 and Section 4.3.

To utilize the different "INFORE Components", the *Streaming Data Engine* and *Online Machine Learning* operators are also added to the streaming extension. They are designed as logical operators as well and can be integrated into executions on all streaming execution backends. For more details about the integration of the "INFORE Components" see the Sections 3.2.5, 3.2.6, 4.5 and 4.6.

If the user intends to optimize his workflow via the Optimizer Component of the INFORE Architecture, he can use the new *Streaming Optimization* operator instead of the *Streaming Nest* operator. For more details see Section 3.2.4 and 4.4.

A streaming analysis workflow designed with the Graphical Editor Component of the INFORE Architecture is depicted in Figure 5. Section 4.2 describes the implementation of the graphical editor component.

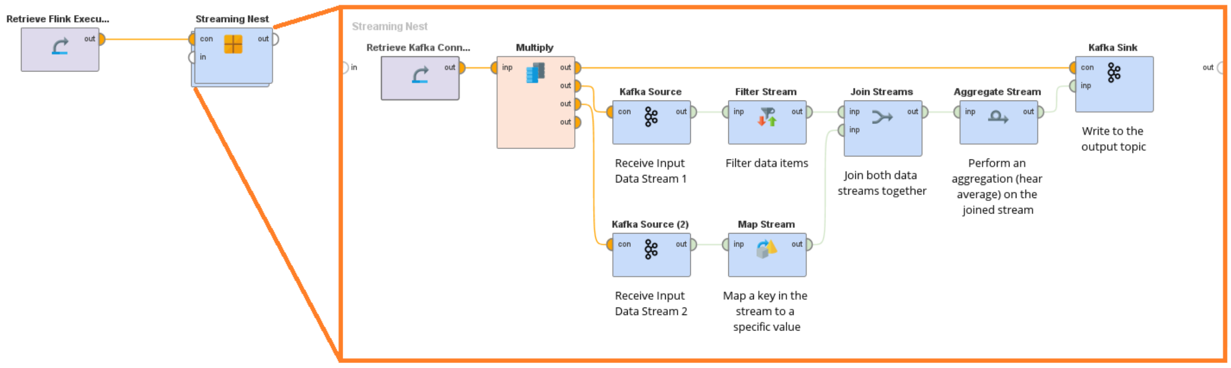


Figure 5: A (demo) streaming analysis workflow designed with the Graphical Editor Component of the INFORE Architecture. All logical streaming analysis operators are placed inside the *Streaming Nest* operator. The Flink Execution Backend connection is provided to the *Streaming Nest* operator and defines where the streaming analysis workflow is deployed.

3.2.3 Manager Component Integration

One of the main principles in the INFORE architecture is to enable a general approach to create streaming analysis workflows that are automatically ported to multiple Big Data streaming backends. The streaming extension delivers this capability to design workflows in a platform-independent fashion, which can be seamlessly executed on any supported streaming backend. This fine engineering feat is achieved by the Manager Component. The user *draws* the visual streaming workflows using the logical operators (explained in Section 3.2.2). Using this visual workflow, the Manager Component builds a generic representation of the streaming workflow called Stream Graph. The Stream Graph is a framework-independent model that can be ported to any supported backend. Currently, these backends include Flink and Spark platforms. This current functionality validates the generic design of the Manager Component as envisioned in the last deliverable D4.1 and beyond proof of concept level.

The Manager Component itself is integrated within the *Streaming Nest* operator, which invokes it upon execution. If the streaming process requires optimization, the user places the entire operator chain inside the *Streaming Optimization* operator (see Section 3.2.4) instead of a *Streaming Nest* operator. Optimization always results in one or more processes, which each start with *Streaming Nest* operator and then on, the Manager Component is invoked in the same fashion by the *Streaming Nest*. Upon invocation, the Manager Component performs various functions and interactions, which are briefly explained (detailed in Section 4.3) below:

- Creating the stream graph from the visual streaming workflow.
- Creating a deployment package that contains a serialized form of stream graph, along with its execution and translation utilities and required (platform-specific) binaries. This package is a self-contained and execution-ready streaming job.
- Dispatching the deployment package to the streaming backend of choice.
- The executable code of the dispatched job recreates the stream graph (at the target platform) by deserializing it and applies the translation utilities (also developed in the project work) to generate a platform-specific graph, which is native to the target platform e.g., Flink or Spark.

From an integration perspective, the Manager Component handles the important link between design phase and execution phase of streaming workflows. To connect with the streaming backends, it uses the connection objects that are provided to the *Streaming Nest* operator as an input (as a collection of connection objects, e.g. to connect with potentially multiple Flink or Spark clusters). Further, the Manager Component provides an extendable programmable approach to implement additional features in the stream graph e.g., to support new sources, sinks, flow-control or transformation functions. This means that for each visual operator of the streaming extension, the relevant programmable hooks of the Manager Component (i.e., classes or methods) are extended or overloaded. This generalization also applies to operators that integrate (either connect or use) other architectural components, which may be distributed as services over different networks (i.e. Synopsis Data Engine (SDE) and Online Machine Learning (OML) components). Thus, on one hand, the functionality provided by these components can be easily integrated within the visual workflows, and on the other hand, during the creation of the stream graph, the usage of these potentially external services (external to the environment where the streaming workflow executes) is also embedded and positioned within the stream graph.

The main outcome of this highly generic and well-integrated design is that end-users can dynamically create varying workflows, without having to hard code any part of the workflow outside RapidMiner Studio. The extensible design allows us to incrementally include as many source, sink, flow-control, transformation or integration operators as required by the use case workflows. Hence, the role of Manager Component is quite broad in achieving the overall integration of INFORE components. For more details on the Manager Component design, please refer to Section 4.3.

3.2.4 Optimizer Component Integration

To integrate the Optimizer Component, the operator *Streaming Optimization* is added to the streaming extension of RapidMiner Studio. It is a subprocess operator, which means that it has an internal subprocess where operators can be placed.

The user designs the logical streaming analysis workflow inside the *Streaming Optimization* operator by utilizing the functionality provided by the Graphical Editor Component (see Section 3.2.2) and by the Synopsis Data Engine Component (see Section 3.2.5) and the Online Machine Learning Component (see Section 3.2.6). Such a designed workflow is shown in Figure 6. The user must also provide a connection to a Kafka cluster to the *Streaming Optimization* operator. This connection is used for the communication between the different execution sites. The user also provides a collection of connections to streaming execution backends on which the workflow can be executed (only on one of them or distributed over several). Both the *Kafka Connection* and the collection of streaming execution backends are handled by the Connection Component (see Section 3.2.1).

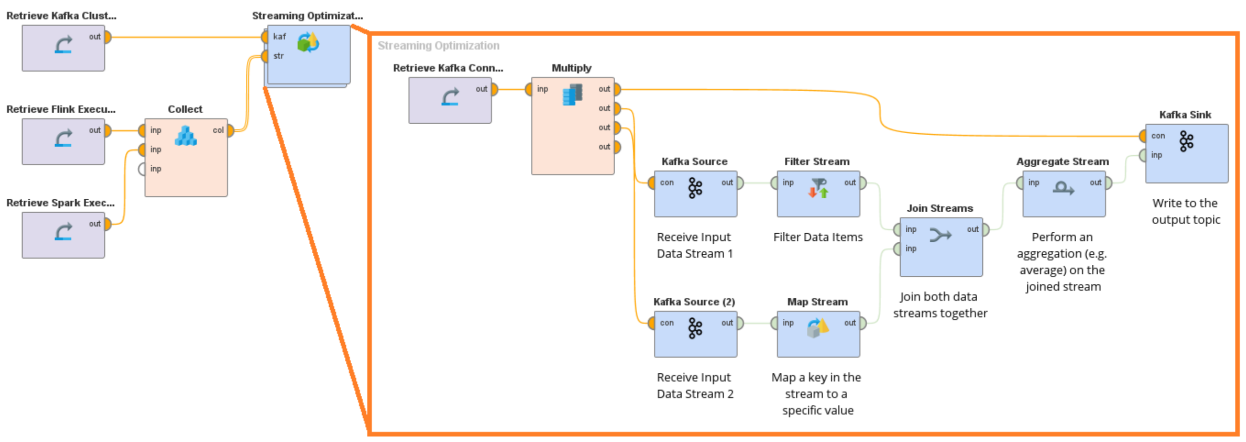


Figure 6: **A streaming analysis workflow in the *Streaming Optimization* operator. The *Kafka Connection* and the collection of the streaming execution backend connections are provided to the *Streaming Optimization* operator as well. They are used by the Optimizer Service to perform the optimization.**

Upon execution of the designed workflow, the *Streaming Optimization* operator starts the Optimizer Service as an in-memory (local) process (see Deliverable D5.1), creates the necessary input JSON files and utilizes the Optimizer Service to optimize the designed workflow. The response of the Optimizer Service contains the placement of the execution of the operators at the provided streaming execution backends.

The *Streaming Optimization* operator receives this response and updates its inner process accordingly. It creates automatically *Streaming Nest* operators and connects the corresponding streaming execution backend connection objects to the input of the *Streaming Nest* operators. If different parts of the workflow are distributed to different streaming execution backends, *Kafka Sink* and *Kafka Source* operators are automatically placed to connect these workflow parts. For this *Kafka Sink* and *Kafka Source* operators, the connection to the Kafka cluster is used, which was provided to the input of the *Streaming Optimization* operator.

The optimized workflow (an example is shown in Figure 7) can be inspected in the inner subprocess of the *Streaming Optimization* operator after the optimization was performed. When the optimized workflow is executed, the different *Streaming Nest* operators deploy their corresponding streaming analysis workflows to the respective streaming execution backends as it is described in Section 3.2.3. The original (logical) workflow can be restored as well.

Section 4.4 describes the implementation of the *Streaming Optimization* operator.

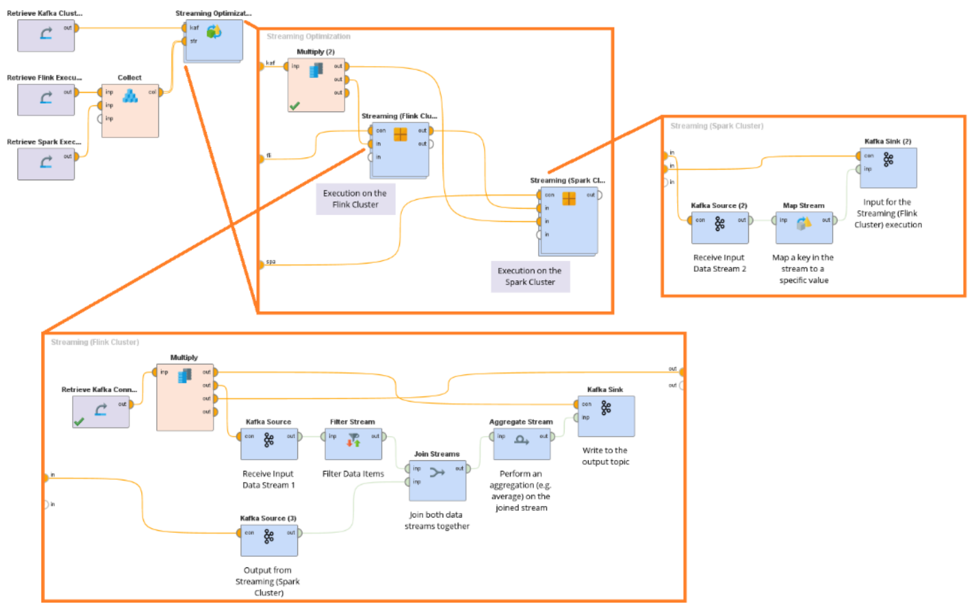
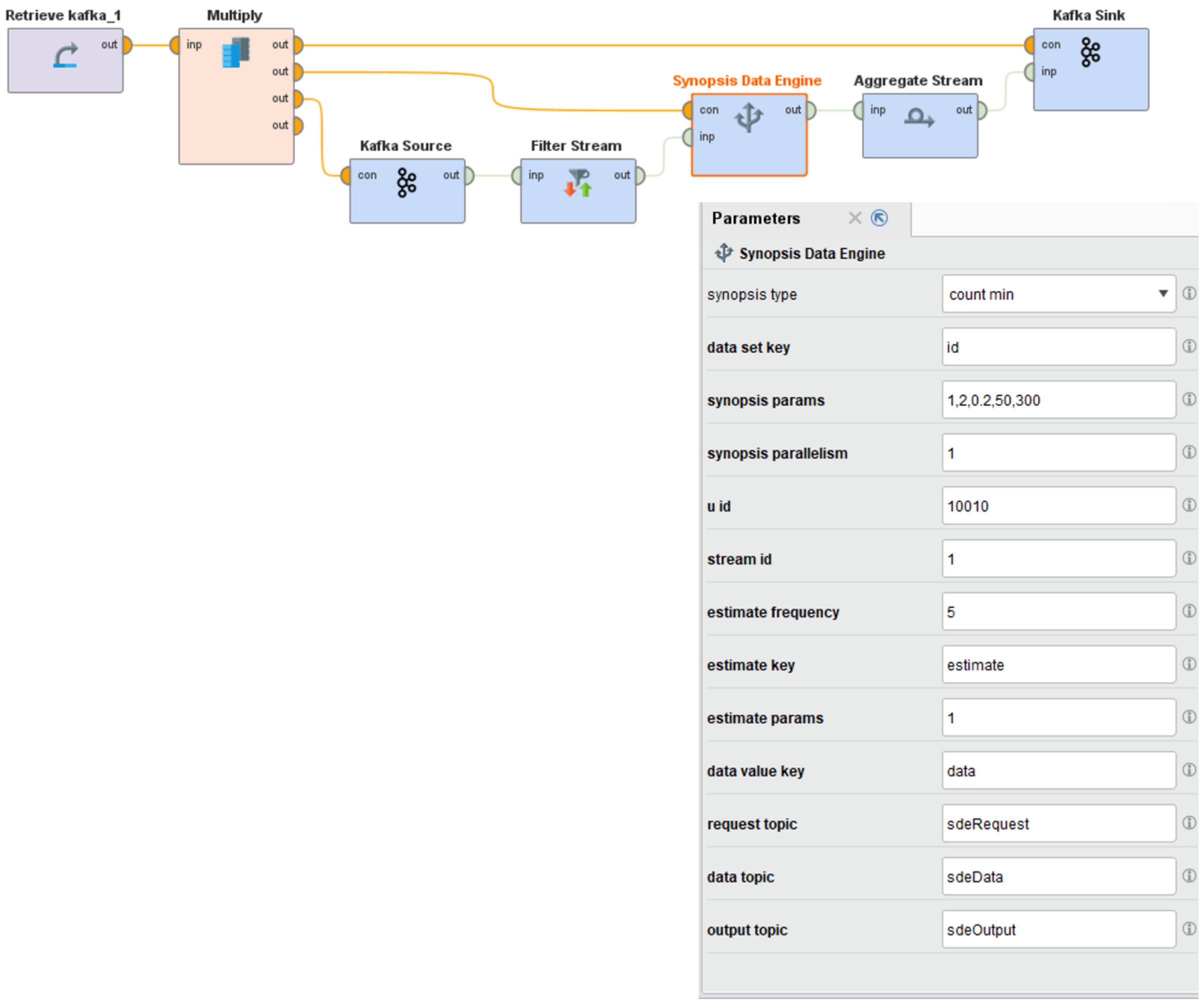

Retrieve Kafka Clust...
Retrieve Flink Execu...
Retrieve Spark Exec...
Collect
Streaming Optimizat...
Streaming Optimization
Multiply (2)
Streaming (Flink Clu...
Execution on the Flink Cluster
Streaming (Spark Cl...
Execution on the Spark Cluster
Streaming (Spark Cluster)
Kafka Sink (2)
Kafka Source (2)
Map Stream
Input for the Streaming (Flink Cluster) execution
Receive Input Data Stream 2
Map a key in the stream to a specific value
Streaming (Flink Cluster)
Multiply
Retrieve Kafka Conn...
Kafka Source
Filter Stream
Join Streams
Aggregate Stream
Kafka Sink
Receive Input Data Stream 1
Filter Data Items
Join both data streams together
Perform an aggregation (e.g. average) on the joined stream
Write to the output topic
Kafka Source (3)
Output from Streaming (Spark Cluster)

Figure 7: The optimized version of the streaming analysis workflow from Figure 6. The logical operators are now placed in the *Streaming Nest* operators. The streaming execution connections are provided to the corresponding *Streaming Nest* operators. The two nests are connected by *Kafka Sink (2)* and *Kafka Source (3)* operators which uses the provided *Kafka Connection* object.

3.2.5 Synopsis Data Engine Component Integration

To integrate the Synopsis Data Engine (SDE) Component, the *Synopsis Data Engine* operator is added to the streaming extension of RapidMiner Studio. It enables users of the INFORE Architecture to utilize the functionality provided by the SDE Service into their streaming analysis, by simply drag and drop the operator into the designed workflow. Figure 8 shows a workflow using the *SDE* operator. The used synopsis and further configurations can be defined by the parameter panel of the operator (bottom right side of Figure 8). The SDE Service is deployed externally and can be used in different workflows. The *Kafka Connection* provided to the input port of the *SDE* operator is used to communicate with the SDE Service. For more details about the SDE component, see Deliverable D5.1. The implementation of the *SDE* operator is described in detail in Section 4.5.



Parameters

Synopsis Data Engine

synopsis type	count min
data set key	id
synopsis params	1,2,0.2,50,300
synopsis parallelism	1
u id	10010
stream id	1
estimate frequency	5
estimate key	estimate
estimate params	1
data value key	data
request topic	sdeRequest
data topic	sdeData
output topic	sdeOutput

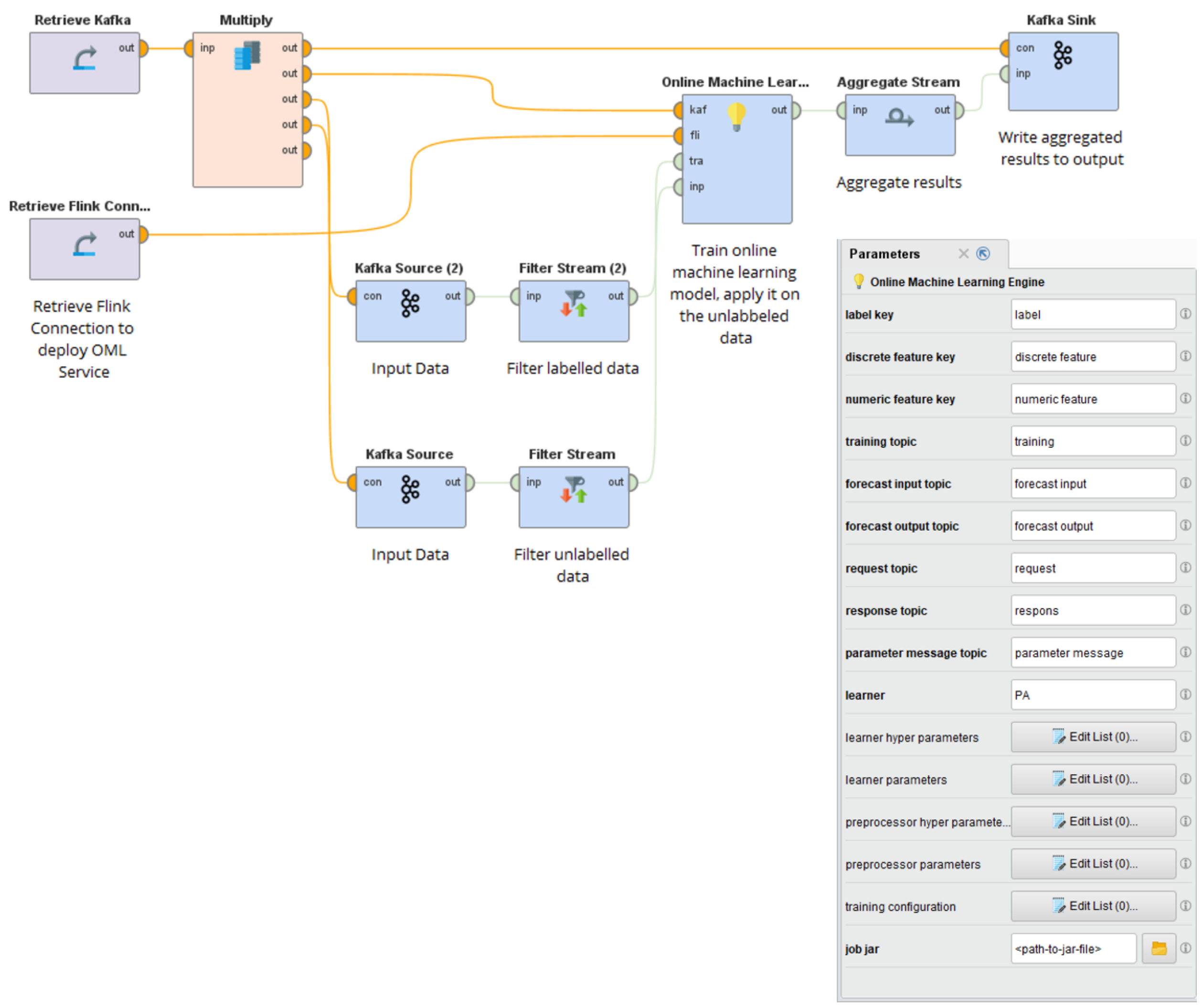
Figure 8: Streaming workflow containing the *Synopsis Data Engine* operator. The parameter panel on the bottom right shows the configuration possibilities for the operator.

3.2.6 Online Machine Learning Component Integration

To integrate the Online Machine Learning (OML) Component, the *Online Machine Learning* operator is added to the streaming extension of RapidMiner Studio. Like the *SDE* operator, the new operator enables users of the INFORE Architecture to include online machine learning in their streaming analysis. The operator is placed in the streaming workflow by drag and drop. It can be configured by the parameter panel. The output is further processed in the streaming analysis workflow.

Figure 9 demonstrates a workflow using the OML operator. The parameter panel of the operator is shown as well. The OML component is not deployed externally as a service like the SDE Service. It is deployed by the *OML* operator itself. To do this, the user must provide a *Flink Connection* to the operator, which defines the Flink cluster, the OML Service is deployed to. The provided *Kafka Connection* is then used to communicate with the new deployed OML Service. The training input data stream and the input data stream to be scored by the trained model must be provided to the corresponding input ports.

For more details about the OML Component see Deliverable D6.2. For the implementation details of the *OML* operator see Section 4.5.



label key	label
discrete feature key	discrete feature
numeric feature key	numeric feature
training topic	training
forecast input topic	forecast input
forecast output topic	forecast output
request topic	request
response topic	respons
parameter message topic	parameter message
learner	PA
learner hyper parameters	Edit List (0)...
learner parameters	Edit List (0)...
preprocessor hyper paramete...	Edit List (0)...
preprocessor parameters	Edit List (0)...
training configuration	Edit List (0)...
job jar	<path-to-jar-file>

Figure 9: Streaming workflow containing the *Online Machine Learning* operator. The parameter panel on the bottom right shows the configuration possibilities for the operator.

3.3 Interactions of the components to build a functional prototype of the INFORE Architecture

Deliverable D4.1 presented a step-by-step process (see Section 8.2.3) on how a streaming analysis workflow is created, optimized and deployed using the INFORE Architecture. This process is also visualized in Figure 10. This subsection describes which steps are implemented by this prototype and which steps will be enhanced in the further development of the implementation of the INFORE Architecture.

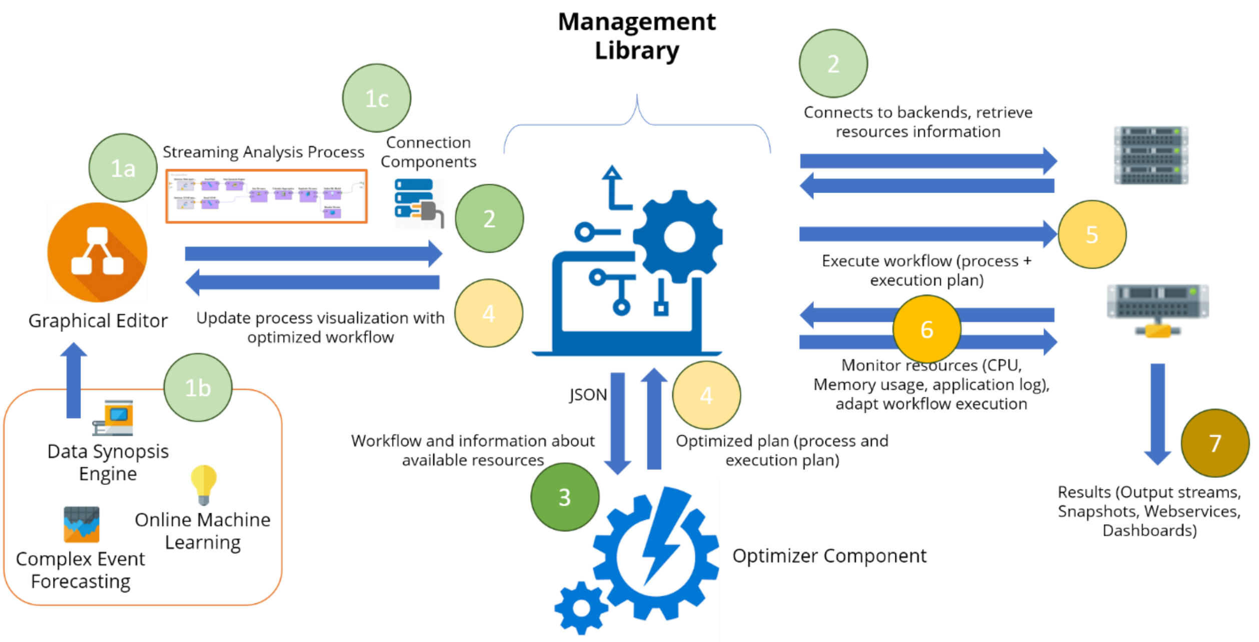

Management Library
1a
1b
1c
2
3
4
5
6
7
Streaming Analysis Process
Connection Components
Graphical Editor
Update process visualization with optimized workflow
Data Synopsis Engine
Online Machine Learning
Complex Event Forecasting
JSON
Workflow and information about available resources
Optimized plan (process and execution plan)
Optimizer Component
Connects to backends, retrieve resources information
Execute workflow (process + execution plan)
Monitor resources (CPU, Memory usage, application log), adapt workflow execution
Results (Output streams, Snapshots, Webservices, Dashboards)

Figure 10: Overview of the step-by-step process for creating and deploying a streaming analysis workflow in the INFORE Architecture (image from deliverable D4.1).

The following list describes the different steps and how the prototype implements these steps:

1) **Design of the streaming analysis workflow:**
The implementation of the Graphical Editor Component (see Section 3.2.2) enables the user to define the logic of a streaming analysis, without the need to care of the technology-specific details.
   a) Streaming operators are implemented as logical operators, abstracting the streaming analysis functionality from the streaming technology providing this functionality. Step 1a is completely implemented by the prototype.
   b) The operators *Synopsis Data Engine* and *Online Machine Learning* (see Section 3.2.5 and 3.2.6) enable us to bring in the functionality of the corresponding components into the streaming analysis workflow. The integration of the Complex Event Forecasting component is ongoing. Step 1b is completely implemented by the prototype for the SDE and OML components.
   c) The implementation of the Connection Component (see Section 3.2.1) enables to create connection objects for input and output streams and for streaming execution backends through the Graphical Editor Component. They can be easily handled by the user in the same drag-and-drop approach as the streaming operators.
   Step 1c is completely implemented by the prototype.

2) **Handover of streaming analysis workflow to Manager Component:**
The implementation of the *Streaming Optimization* operator (see Section 3.2.4) can create all necessary JSON representations of the designed workflow and of the available streaming execution backends, while the *Streaming Nest* operator (see Section 3.2.3) can create a stream graph representation of the designed workflow. This functionality achieves the desired handover of the streaming analysis workflow to the Manager Component.

   While the connection objects can be utilized to connect to the input and output streams and the streaming execution backends (which is used to deploy the workflows on the computing clusters), the information retrieval of available resources is not yet integrated fully in the INFORE Architecture. The main objectives of Step 2 are implemented by the prototype, while there will be ongoing development on the information retrieval.

3) **Optimization of the streaming analysis workflow by the Optimizer Component:** The implementation of the Optimizer Component (see Section 3.2.4 and deliverable D5.1) achieves the possibility to convert the created JSON representations of the workflows and available computing clusters (see step 2) into a tool-agnostic workflow representation which can be optimized. While the optimization includes the bundling of operators to specific streaming execution jobs (and thereby the placement of the operators on the computing clusters), additional optimization efforts are not yet integrated in the INFORE Architecture.

   The main functionality of Step 3, the optimization of a streaming analysis workflow provided by the Manager Component, is implemented by the prototype, while further optimization possibilities will be enhanced in future developments.

4) **Providing optimized workflow back to Manager Component:**
   The implementation of the *Streaming Optimization* operator (see Section 3.2.4) can receive the response of the Optimizer Component. The *Streaming Optimization* operator automatically creates different *Streaming Nest* operators, depending on the response of the Optimizer Component. These different *Streaming Nest* operators can be visualized by the Graphical Editor, informing the user about the changes of the Optimizer Component. The *Streaming Nest* operators can create Stream Graph representations of the cluster specific streaming jobs and hence the Manager Component is updated on the optimized workflow. Step 4 is completely implemented by the prototype.

5) **Deployment of the workflow by the Manager Component:**
   The implementation of the Manager Component (see Section 3.2.3) can create technology specific jobs from the Stream Graph representations of the workflows and deploy them on the corresponding computing cluster.

   Step 5 is implemented for the streaming technologies Flink and Spark by the prototype.

6) **Monitoring and management of the deployed workflow:**
   The monitoring and management of the deployed workflow is not yet implemented in this prototype of the INFORE Architecture. Users must monitor, abort or pause running workflows manually. They can still use the already implemented functionality of steps 1) to 5) to update existing workflows, perform a re-optimization and re-deploy them on the computing clusters.

   Step 6 is not yet implemented by the prototype. But the steps 1) to 5) enable an easy manual version of Step 6.

7) **Retrieving and consuming results:**
   The implementation of the *Kafka Sink* operator (see Section 3.2.2) enables to create output streams, which can be further consumed by different streaming consumers. Other means of providing results of streaming analysis workflows are not yet implemented by this prototype.

   The main functionality of Step 7 is implemented by the prototype, while further means of providing results will be added in future developments.

To summarize, the prototype of the INFORE Architecture, described in this deliverable achieve the implementation of the main functionality of all steps. Hence, it is a functional prototype of the architecture design, which was described in deliverable D4.1.

4 Specification and Design of INFORE Software Stacks

This section details the implementations of the different components in the streaming extension of RapidMiner Studio. Hence, the implementations of the Connection Component, the Graphical Editor Component and the Manager Component are described. In addition, the implementations of the integrations of the Optimizer Component, the Synopsis Data Engine Component and the Online Machine Learning Component are described in this Section. The details of the implementations of the components themselves are described in the respective deliverables. A first discussion about the realization of the INFORE Architecture was already presented in deliverable D4.1. The following implementation details are based on this discussion, hence the Section 5 of the deliverable D4.1 is added to the appendix of this document (see Section 8.2) for convenience.

4.1 Connection Component

The new Connection Component of the INFORE Architecture extends the already existing connection object management of RapidMiner[2]. It adds three new connection objects types which are utilized to connect to the corresponding streaming backends. Section 3.2.1 describes the usage of the new connection objects in the INFORE Architecture. Following the settings of each new connection type is listed. While the current list of settings is enough to achieve the goals of the prototype of the INFORE Architecture, the connection object types will probably be enhanced in the future while finalizing the prototype in the upcoming development in the INFORE research project.

Kafka Connection: The *Kafka Connection* objects are utilized by the *Kafka Source*, *Kafka Sink*, *Synopsis Data Engine* and the *Online Machine Learning* operators to receive streamed data from Kafka topics or to produce streamed data to Kafka topics on a Kafka Cluster. This can be used for input and output data in the streaming analysis workflow, as well as the middleware communication between the different components of the INFORE Architecture.

To create a *Kafka Connection*, a *comma separated lists of hosts and ports* must be provided. This host and port combinations are the connection points to a running Kafka Cluster. The topic names from which data is received or to which data is produced are configuration parameters of the corresponding operators and are not provided in the connection object. Figure 2 shows the creation of a *Kafka Connection* object.

Flink Connection: The *Flink Connection* objects are utilized by the *Streaming Nest* operator to deploy a logical streaming analysis workflow on a Flink cluster. Like for the *Kafka Connection*, the user must specify the *host* and *port* used to connect to a running Flink Cluster. In addition, the user must specify the *parallelism* used for submitted Flink jobs on this cluster. Figure 3 shows the creation of a *Flink Connection* object.

Spark Connection: Like the *Flink Connection* objects, *Spark Connection* objects are used by the *Streaming Nest* operator to deploy logical streaming analysis workflows on Spark clusters. Upon creation of a *Spark Connection* object, the user must specify the name of the Spark *Master* node to connect to. In addition, he must specify the *URI* to the HDFS (cluster) path where the artefacts (job execution package, serialized stream graph) will be deposited and thus made available for jobs running on Spark.

Figure 4 shows the creation of a *Spark Connection* object.

4.2 Graphical Editor Component

The Graphical Editor Component builds on the graphical user interface of RapidMiner Studio. The streaming extension adds new operators, which provides the functionality to design and deploy a streaming analysis workflow in a non-coding way by a simple-to-use drag & drop approach. The main two new operators are the *Streaming Nest* operator and the *Streaming Optimization* operator.

Streaming Nest: The *Streaming Nest* operator is a subprocess operator. A workflow designed in its inner subprocess represents a single streaming analysis job on a streaming execution backend. The connection object

[2] https://docs.rapidminer.com/latest/studio/connect/

provided at the *"connection"* input port defines which backend will be used. Any RapidMiner object (e.g. connection objects used in the streaming analysis workflow) can be passed in and taken from the inner subprocess of the *Streaming Nest* operator by using the corresponding *"in"* and *"out"* throughput ports. The *"job name"* parameter specifies the name of streaming job which will be deployed by the *Streaming Nest* operator to the streaming execution backend. For deploying streaming analysis workflows, the *Streaming Nest* operator uses the Manager Component, which is described in Section 3.2.3 and 4.3.

Streaming Optimization: Like the *Streaming Nest* operator, the *Streaming Optimization* operator is a subprocess operator. A logical streaming analysis workflow is designed by the user in its subprocess. In addition, the user provides a *Kafka Connection* object to the *"kafka connection"* input port of the *Streaming Optimization* operator. Also, a collection (e.g. by using the *Collect* operator) of streaming execution connection objects are provided to the *"streaming execution connections"* input port of the operator. The *Streaming Optimization* operator uses the Optimization Component, to optimize the designed workflow. The parameter *"optimizer directory"* defines where the Optimizer Component is installed. The collection of streaming execution connections is used as the possible execution backends for the optimization. The *Kafka Connection* is used to connect different parts of the workflow in case the execution is distributed to different backends. The button *"Restore Process"* can be used to restore the original logical streaming analysis workflow. For more details about the optimization component and its integration see Sections 3.2.4 and 4.4.

The logical streaming operators of the streaming extension can be grouped into different subgroups.
Stream In- and Output Operators:

- Kafka Source:
    - Enables to receive streamed data from a Kafka topic.
    - A *Kafka Connection* is used to connect to the Kafka Cluster the parameter *"topic"* specifies the name of the Kafka topic from which data is received.
- Kafka Sink:
    - Enables to write streamed data to a Kafka topic.
    - A *Kafka Connection* is used to connect to the Kafka Cluster the parameter *"topic"* specifies the name of the Kafka topic to which data is written.

Stream Transformation Operators:

- Aggregate Stream:
    - Enables to calculate aggregates (sum, average, count, min, max) of values in streamed data over a time window.
    - The parameter *"key"* specifies the partitioning of the data, the parameter *"value key"* specify the value, which is aggregated, the parameter *"window length"* the length of the time window and the *"function"* specifies the aggregation function.
- Filter Stream:
    - Enables to filter streamed data items.
    - The parameter *"key"* specifies the key of the value to be evaluated in the filter, the parameter *"value"* specifies the operand of the filter predicate and the parameter *"operator"* specifies the filter operation.
- Map Stream:
    - Enables to map a field in streamed data to a new value.
    - The parameter *"key"* specifies the key, which is mapped, while the parameter *"value"* specifies the new value of the field.

Stream Flow Control Operators:

- Duplicate Streams:
    - Enables the duplication of a data stream.
    - The two output stream ports represent the duplicated data streams.
- Join Streams:
    - Enables to join two data streams.
    - The two streams must be provided to the input ports. The parameters *"left key"* and *"right key"* specify the keys in the left/right data stream to perform the join. The parameter *"window length"* specifies the time window within the join is performed.

- Connect Streams:
  - Enables to connect to data streams.
  - The control stream and the data stream must be provided to the input ports. The parameters *"control key"* and *"data key"* specify the keys used in the corresponding streams.

INFORE Component Operators:

- Synopsis Data Engine:
  - Enables to leverage the functionality of the Synopsis Data Engine component in a streaming analysis workflow
  - A *Kafka Connection*, which is used to connect to the Synopsis Data Engine Component, must be provided at the input port. Several parameters configure the used Synopsis. For more details see Section 3.2.5 and 4.5.
- Online Machine Learning:
  - Enables to leverage the functionality of the Online Machine Learning component in a streaming analysis workflow
  - A *Kafka Connection*, which is used to connect to the Online Machine Learning Component, must be provided at the input port. Several parameters configure the used Synopsis. For more details see Section 3.2.6 and 4.6.

4.3 Manager Component

The Manager Component is designed as a set of extensible programmable modules, which together offer a comprehensive functionality as outlined in Section 3.2.3. In this section, we present these modules and show how the multi-layered translation of the streaming workflows is achieved by these modules; starting from visual design to a framework-independent representation (stream graph) and further downstream into a platform-specific translation.

The high-level design of Manager Component can be expressed in terms of its multi-layered functionality, as shown in Figure 11.

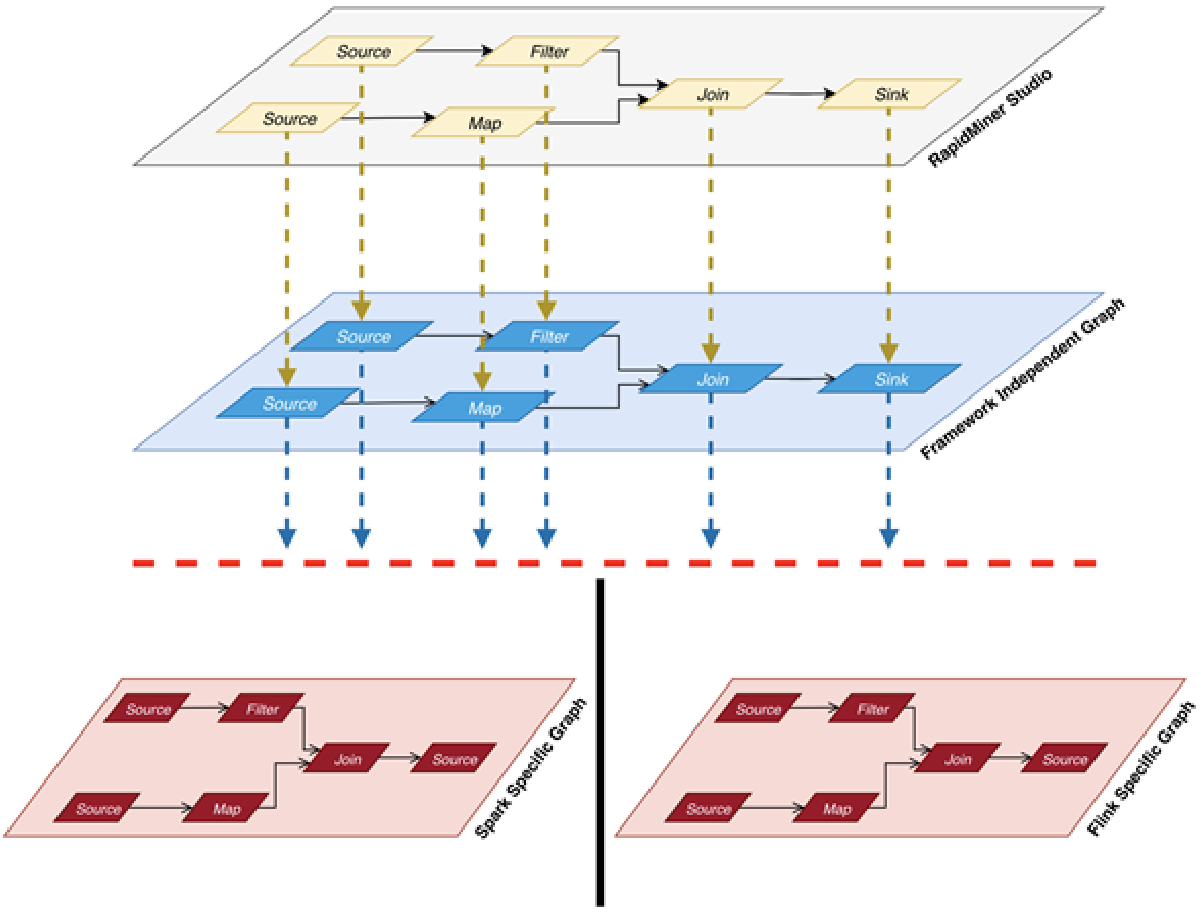


Figure 11: High-level design of Manager Component can be expressed in terms of its multi-layered functionality.

Extendable and Programmable Module for Graph Creation:

The Manager Component provides programmable interfaces to convert each new visual operator into a node of the stream graph. Upon execution of process (within RapidMiner Studio), the stream graph is built progressively and as per the sequential execution order of the visual operators in the operator chain. The stream graph gets extended with each next operator in operator chain, which may be source, sink, flow-control, transformation or integration operators. Under the hood, the programmable interface for each visual operator (consider a transformation operator for instance), is composed of a Transformer class, which is instantiated to represent a node in the stream graph. An instance of the Transformer class encapsulates all consuming (incoming/upstream) and producing (outgoing/downstream) streams that affect the node, which the transformer instance represents. Some additional framework-level classes have been written, which serve as base classes to be extended for each visual operator and some standard methods need to be overloaded, so that the stream graph could be processed using a common scheme for different operators.

Once the stream graph is created, it is serialized and packaged along with its executing code into a so-called fat-jar file, which also contains the platform-specific binaries as well translation utilities. There are a few subtleties when it comes to preparing and dispatching the packages targeting the underlying platforms e.g., the platform-specific binaries that accompany the streaming graph may be separately posted for Spark, but in any case, the delivery of the streaming graph and its dependencies is ensured before the streaming graph is executed. These rather low-level integration mechanisms have been resolved considering the most feasible approaches available in individual platforms, with the result being that the end-user is not affected and can rely on seamless dispatching of her streaming graph. Once received and assembled at the backend, the streaming graph is executed. The executing code is basically a platform-specific program (a Job in case of Flink or Spark) that first, deserializes the stream graph on the target platform and then performs platform-specific translations on the stream graph. These translations are taken care by the Translation Utilities module.

The transformation of the visual workflow into a generic stream graph, its dispatchment as a deployment package and subsequent translation into platform specific stream graph executed as a native job is sketched out in the Figure 12 below.

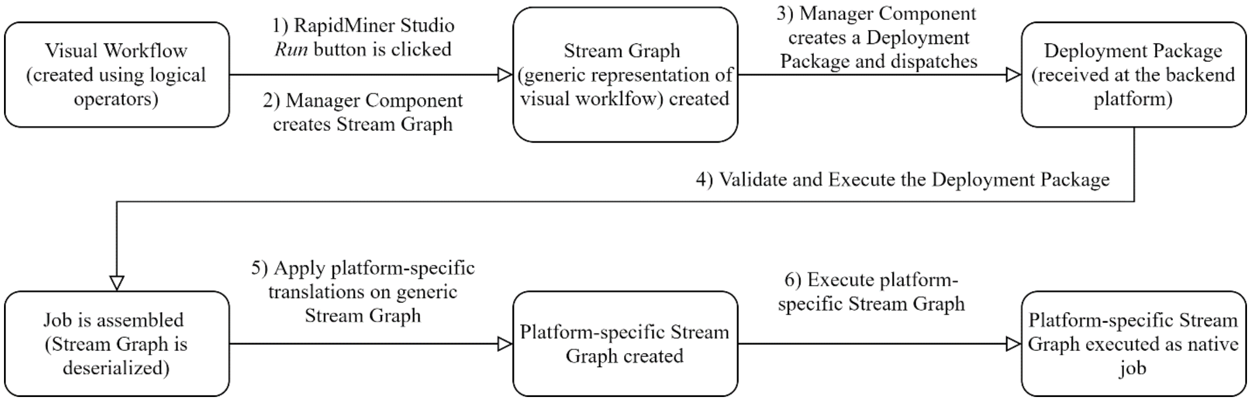

Visual Workflow (created using logical operators)
1) RapidMiner Studio *Run* button is clicked
2) Manager Component creates Stream Graph
Stream Graph (generic representation of visual worklfow) created
3) Manager Component creates a Deployment Package and dispatches
Deployment Package (received at the backend platform)
4) Validate and Execute the Deployment Package
Job is assembled (Stream Graph is deserialized)
5) Apply platform-specific translations on generic Stream Graph
Platform-specific Stream Graph created
6) Execute platform-specific Stream Graph
Platform-specific Stream Graph executed as native job

Figure 12: Various stages in the transformation, deployment and execution of workflows – represented here as states (rectangles) and actions (numbered sequences) to progress to next state.

Translation Utilities Module:

A platform-specific Translation Manager utility (class) traverses the stream graph object model to convert it into a streaming program (or Job) that can be executed on the said platform. Traversal is performed using the Visitor design pattern. As each node of the stream graph represents a unique function, hence, for each operator, a new *visit* method needs to be overloaded in the Translation Manager class, with a new parameter representing the transformer instance. The *visit* method provides the hook where the generic representation of the operator is converted into the target platform's representation. Inside the *visit* method, the platform specific libraries (also shipped with the fat jar), are utilized to perform sanity tests e.g. the existence or availability of all incoming and outgoing streams on the target environment and then the transformer functionality is translated into natively compatible streaming functions.

Remember that the Translation Manager utilities are platform-specific, so there are separate Translation Manager(s) for each backend, which are also structured in different project-modules, but within the codebase of the streaming extension. The Translation Manager for Flink for instance, would require a library of so-called Translator classes - each shadowing a unique (but generic) transformer or another operator. The Translator classes must implement their platform-specific logic on stream handling, transformation or more, within a *translate* method, which is invoked by the Translation Manager.

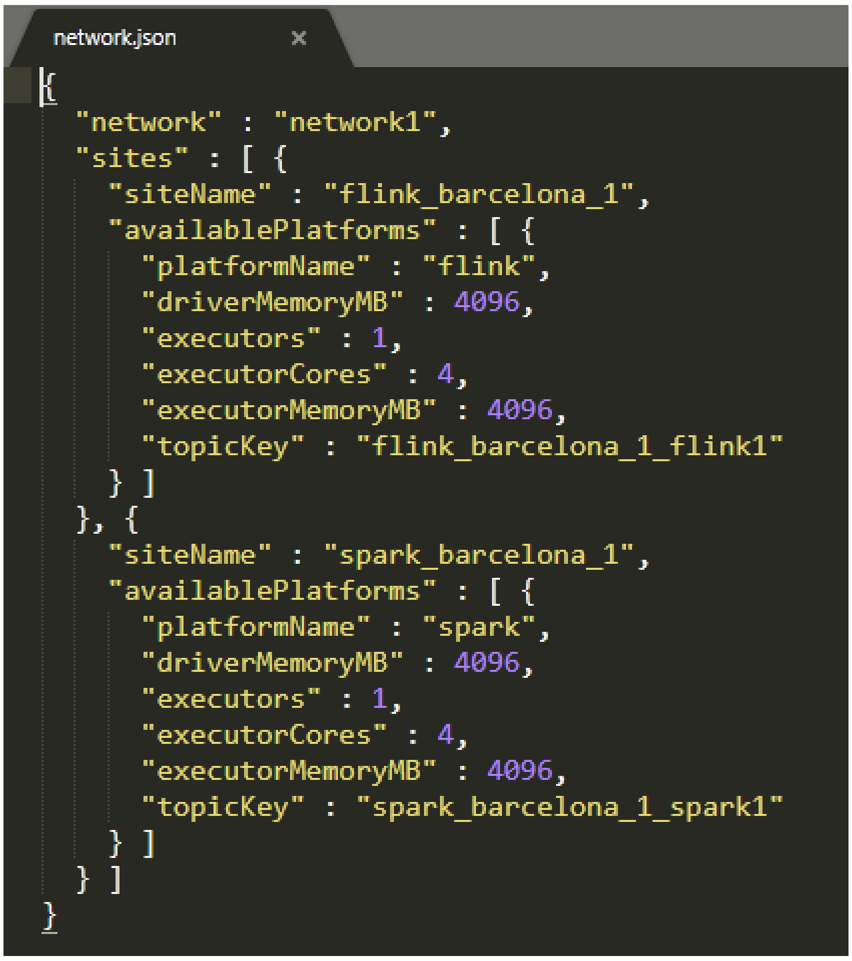
As an example, let us consider that we need to create a new transformation operator (say *Connect Streams* operator) to connect two streams. For this, the developer implements a ConnectTransformer class (for the generic model), then adds a visit method in the TranslationManager classes – separately for Flink and Spark and finally, to achieve the translation into native platform (within the *visit* method), the ConnectTransformerTranslator class has to be implemented for both Flink and Spark Translation Managers and provides respective implementations of the *translate* method in each. The translate method acts as the place-holder, where the libraries and APIs specific to the target platform are used to provide the desired functionality e.g., in our example case, the DataStream class from Flink (and Spark) can now be used to read two different streams incoming in the *Connect Streams* operator and connect them on the user-provided key parameter.

This standard nomenclature for model conversion has been tested to satisfaction with the first batch of operators in the streaming extension. The design has eased development and is expected to ease management effort over time, while also affording pluggability of operators as the number of streaming operators would increase in accordance with and subject to the use case requirements.

4.4 Integration of the Optimizer Component

The *Streaming Optimization* operator is added to the streaming extension of RapidMiner to integrate the Optimizer Component. The operator has two input ports. A *Kafka Connection* object must be connected to the first *"kafka connection"* input port. A collection of streaming execution connection objects (e.g. *Flink* and/or *Spark Connections*) must be provided to the second input port (*"streaming execution connections"*). The user defines a streaming analysis workflow in the inner subprocess of the operator. The user can drag and drop logical streaming operators provided by the Graphical Editor Component (see Section 4.2) in the subprocess.
Upon execution of the operator the Optimizer Service is started as an in-memory (local) process. To do this the parameter *"optimizer directory"* specifies the location of the Optimizer. The operator creates then the input JSON objects for the Optimizer:

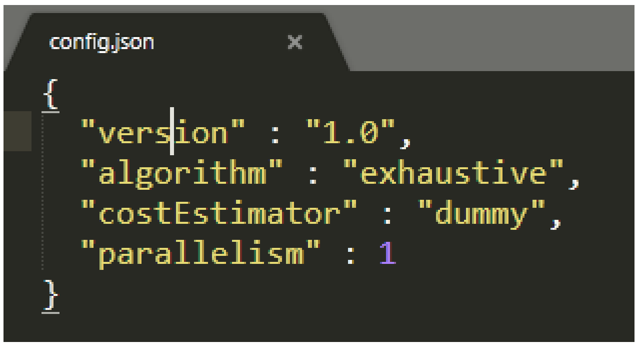
Network.json: The Network JSON describes the available execution sites and platforms. The collection of streaming execution connection objects is used to create the Network JSON. Each connection is used to create an entry of one platform on one site in the Network JSON. For this prototype, resource variables in the JSON objects are set to dummy values. Figure 13 shows an example Network JSON object.



```
{
  "network" : "network1",
  "sites" : [ {
    "siteName" : "flink_barcelona_1",
    "availablePlatforms" : [ {
      "platformName" : "flink",
      "driverMemoryMB" : 4096,
      "executors" : 1,
      "executorCores" : 4,
      "executorMemoryMB" : 4096,
      "topicKey" : "flink_barcelona_1_flink1"
    } ]
  }, {
    "siteName" : "spark_barcelona_1",
    "availablePlatforms" : [ {
      "platformName" : "spark",
      "driverMemoryMB" : 4096,
      "executors" : 1,
      "executorCores" : 4,
      "executorMemoryMB" : 4096,
      "topicKey" : "spark_barcelona_1_spark1"
    } ]
  } ]
}
```

Figure 13: Example Network JSON objects for communication with the Optimizer Component. Each connection object is represented by a *site-platform* entry. Resource variables are set to dummy values for this prototype.

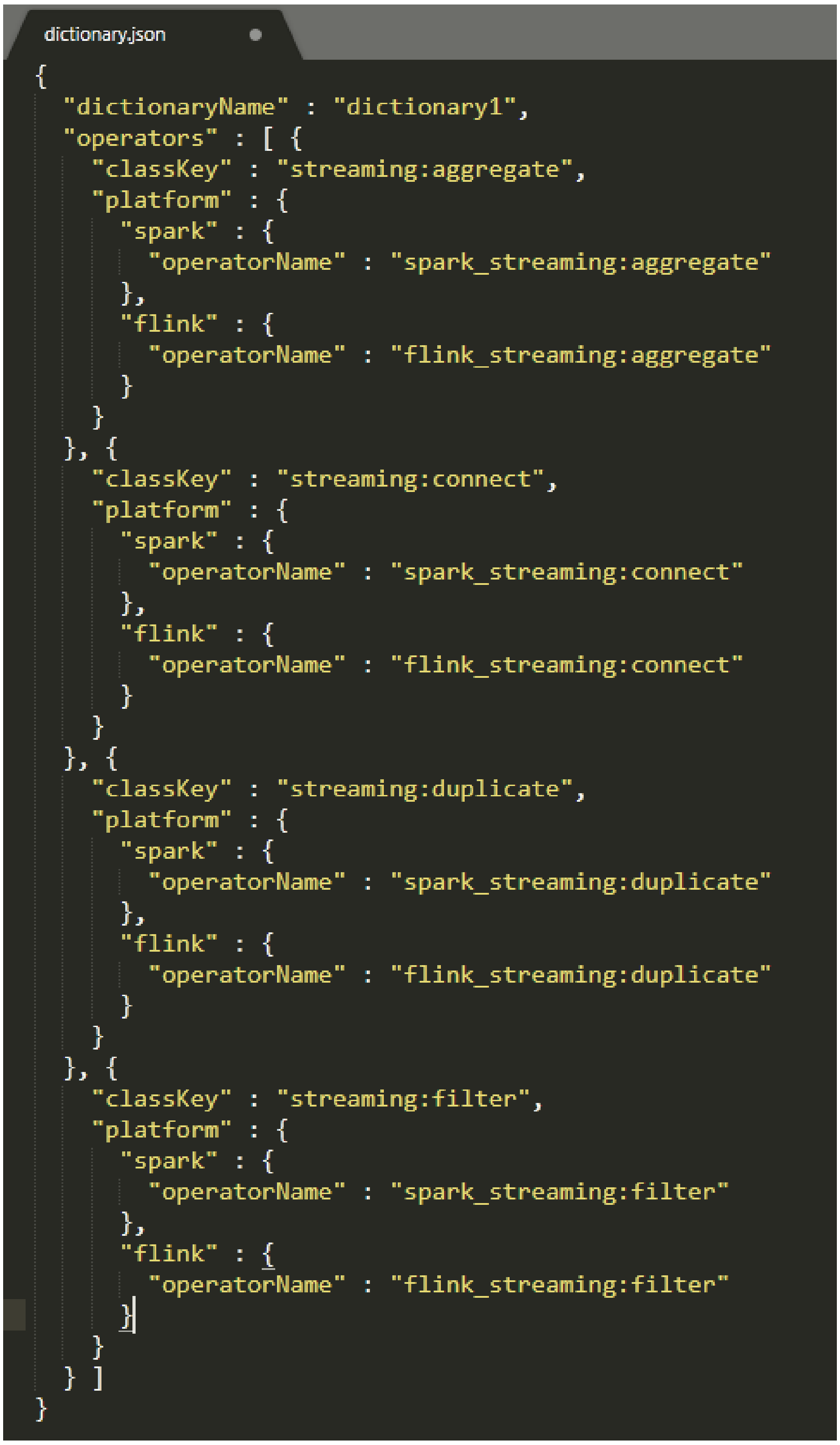
Config.json: The Config JSON describes the setting of the optimizer algorithms. For this prototype, default settings are used, without the possibility to change the settings by the user. This will be enhanced in the future. Figure 14 shows an example Config JSON object.



```
{
    "version" : "1.0",
    "algorithm" : "exhaustive",
    "costEstimator" : "dummy",
    "parallelism" : 1
}
```

Figure 14: **Example Config JSON object for communication with the Optimizer Component. For this prototype default settings are used, without the possibility to change the settings by the user.**

Dictionary.json: The Dictionary JSON describes the available logical streaming operators. For this prototype, all streaming operators of the Graphical Editor Component (see Section 4.2) are listed in the Dictionary JSON. As described in Section 3.1 streaming operators are all designed as logical operators, while the physical implementation is defined by the connection object provided to the *Streaming Nest* operator. Thus, the field *"platform"* in the Dictionary JSON object lists the supported technologies for the operators by providing dummy *"operatorName"* fields for the different technologies. For this prototype all operators support Flink and Spark. Figure 15 shows an example Dictionary JSON object.



dictionary.json

```
{
  "dictionaryName" : "dictionary1",
  "operators" : [ {
    "classKey" : "streaming:aggregate",
    "platform" : {
      "spark" : {
        "operatorName" : "spark_streaming:aggregate"
      },
      "flink" : {
        "operatorName" : "flink_streaming:aggregate"
      }
    }
  }, {
    "classKey" : "streaming:connect",
    "platform" : {
      "spark" : {
        "operatorName" : "spark_streaming:connect"
      },
      "flink" : {
        "operatorName" : "flink_streaming:connect"
      }
    }
  }, {
    "classKey" : "streaming:duplicate",
    "platform" : {
      "spark" : {
        "operatorName" : "spark_streaming:duplicate"
      },
      "flink" : {
        "operatorName" : "flink_streaming:duplicate"
      }
    }
  }, {
    "classKey" : "streaming:filter",
    "platform" : {
      "spark" : {
        "operatorName" : "spark_streaming:filter"
      },
      "flink" : {
        "operatorName" : "flink_streaming:filter"
      }
    }
  } ]
}
```

Figure 15: Example Dictionary JSON object for communication with the Optimizer Component. For demonstration purpose, the list of supported streaming operators is shortened in this figure. The "*operatorName*" fields contain dummy entries, only indicating that the operator supports the corresponding platform.

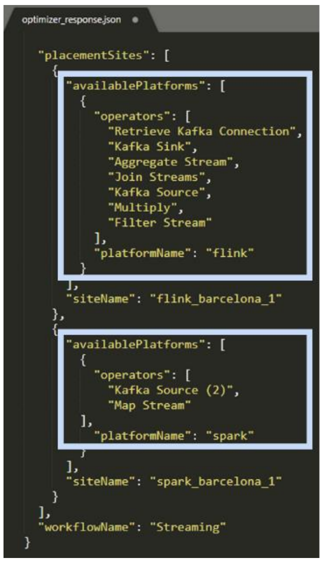
Workflow.json: The Workflow JSON describes the designed streaming analysis workflow in the subprocess of the *Streaming Optimization* operator. For each operator it contains information about the name, the parameters, the input and output ports and the operator class (e.g. its logical functionality). In addition, all connections between operators are provided in the workflow JSON. deliverable D4.1 already contained a description of the Workflow JSON, as the description of the interface between Manager Component and Optimizer Component (see Section 8.2.1, Table 2 and Figure 28). While developing the prototype of the INFORE Architecture, the implementation of the streaming operator as well as the integration concepts of the Optimizer Component were concretized (see Section 3.1). This leads to an update of the fields in the engine-agnostic workflow representation in comparison to deliverable D4.1. Table 1 gives an overview about the updated fields and their meaning in the representation. Most notable is the removal of the "*isLogicalOperator"* field, cause all operators are logical ones. Also, the *"Resources"* information fields are moved to the Network JSON object. Other changes address necessary fields to correctly interpret the representation of the workflow, which came up during the development of the integration of the Optimizer Component.

Key	Description	Key	Description
Workflow Information		**numberOfSubprocesses**	Number of subprocesses (can be null)
workflowName	Name of the current workflow	**innerWorkflows**	List of inner workflows (each field contains a complete workflow representation, see *Workflow Information*) (can be null)
enclosingOperatorName	Name of enclosing operator of this workflow	**Parameter**	
innerSourcesPorts-AndSchemas	List of inner sources of the current workflow (fields of each entry described in *PortAndSchema*)	**key**	Key of the parameter
innerSinksPorts-AndSchemas	List of inner sinks of the current workflow (fields of each entry described in *PortAndSchema*)	**value**	Current parameter value (as String)
operatorConnections	List of *Operator Connections* (fields of each entry described in *Operator Connections*) in the current workflow	**defaultValue**	Default value (as String)
operators	List of *Operators* (fields of each entry described in *Operator*) in the current workflow	**range**	String representation of the range
Operator Connections		**typeClass**	Name of the Java parameter class
fromOperator	Name of source operator	**PortAndSchema**	
fromPort	Name of source port	**name**	Name of the input or output port
fromPortType	Type (normal output port or inner source) of source port	**objectClass**	Name of java class of the object delivered or received at this port
toOperator	Name of sink operator	**portType**	Type (output port, input port, inner source, inner sink) of the port
toPort	Name of sink port	**isConnected**	Indicator if the port is connected
toPortType	Type (normal input port or inner sink) of sink port	**schema**	If object is a data set, *Schema* of this data set
Operator		**Schema**	

name	(unique) name of the operator	**fromMetaData**	Indicator if schema is retrieved from meta data
classKey	RM specific key for the operator class	**size**	Size of the data set at the port
operatorClass	Name of the java operator class	**attributes**	List of *Attributes*
isEnabled	Indicator if the operator is enabled	**Attribute**	
parameters	List of Parameters (see *Parameter*)	**name**	Name of Attribute
inputPortsAndSchemas	List of input ports and their schema (see *PortAndSchema*)	**type**	Value type
outputPortsAndSchemas	List of output ports and their schema (see *PortAndSchema*)	**specialRole**	Special role
hasSubprocesses	Indicator if the operator has subprocesses		

Table 1: Updated overview of the different fields and their meaning in the engine-agnostic workflow representation.

The created JSON objects are send with HTTP requests to the Optimizer Service. The information is used by the Optimizer Service to perform an optimization of the provided Workflow JSON. The response of the Optimizer Service is the Workflow JSON with updated information about the placement of the operators on the different streaming execution backends. Figure 16 shows the response JSON of the Optimizer Service and a visualization of the placement of the operators. It is also indicated that two operator connections in the workflow will be split by the placement of the operator.


optimizer_response.json

```
  "placementSites": [
    {
      "availablePlatforms": [
        {
          "operators": [
            "Retrieve Kafka Connection",
            "Kafka Sink",
            "Aggregate Stream",
            "Join Streams",
            "Kafka Source",
            "Multiply",
            "Filter Stream"
          ],
          "platformName": "flink"
        }
      ],
      "siteName": "flink_barcelona_1"
    },
    {
      "availablePlatforms": [
        {
          "operators": [
            "Kafka Source (2)",
            "Map Stream"
          ],
          "platformName": "spark"
        }
      ],
      "siteName": "spark_barcelona_1"
    }
  ],
  "workflowName": "Streaming"
}
```

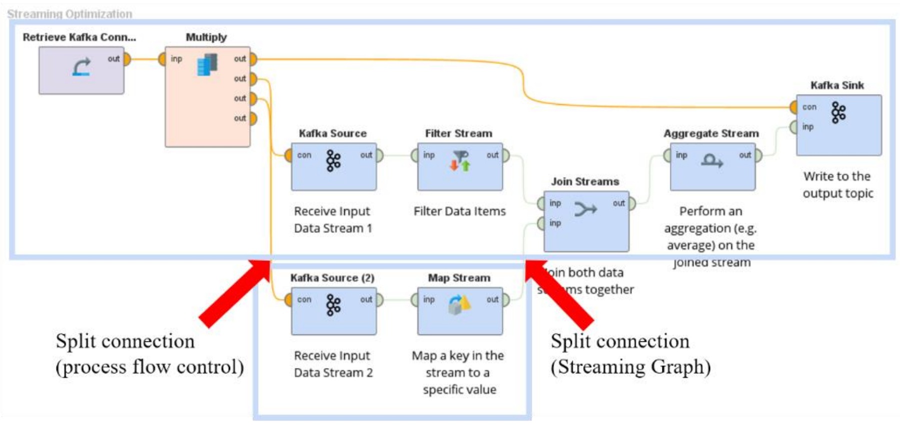

Streaming Optimization
Retrieve Kafka Conn...
out
Multiply
inp
out
out
out
out
Kafka Source
con
out
Receive Input Data Stream 1
Filter Stream
inp
out
Filter Data Items
Join Streams
inp
inp
out
Join both data streams together
Aggregate Stream
inp
out
Perform an aggregation (e.g. average) on the joined stream
Kafka Sink
con
inp
Write to the output topic
Kafka Source (2)
con
out
Receive Input Data Stream 2
Map Stream
inp
out
Map a key in the stream to a specific value
Split connection (process flow control)
Split connection (Streaming Graph)

Figure 16: Response JSON from the Optimizer Service, containing updated information about the placement of the operators on the different streaming execution backends. This placement is also visualized for the designed example workflow. Two operator connections will be split by this placement.

The *Streaming Optimization* operator creates now automatically *Streaming Nest* operators for all streaming execution backends on which at least one operator is placed on. The corresponding streaming execution connection is connected to the *Streaming Nest* operators. This is shown in the top of Figure 17.

If parts of the Streaming Analysis Workflow are placed on different streaming execution backends, which is the case for the connection between the operators *Map Stream* and *Join Streams* in the workflow shown in Figure 17, the parts must be connected by the Kafka Middleware. To achieve this, *Kafka Sink* and *Kafka Source* operators are automatically placed at the now split operator connections. The name of the Kafka topic used is constructed as follows:

<SourceOp>.<SourcePort> to <TargetOp>.<TargetPort>

SourceOp: Name of the source operator of the original split operator connection
SourcePort: Name of the source port of the original split operator connection
TargetOp: Name of the target operator of the original split operator connection
TargetPort: Name of the target port of the original split operator connection

The topic parameters of the inserted *Kafka Sink* and *Kafka Source* operator are set accordingly. The *Kafka Connection* object from the "k*afka connection"* input port of the *Streaming Optimization* operator is connected to the *Kafka Sink* and *Kafka Source* operators. This Stream connection based on Kafka between the different streaming execution backends is symbolized by the red dashed line (small dashes) in Figure 17.

If an operator connection, which was used for process flow control and which was not part of the stream graph, is split between different *Streaming Nests,* the connection is re-established by using the *"in"* and *"out"* throughput ports of the *Streaming Nest* operators. This is symbolized by the red dashed line (larger dashes) in Figure 17.

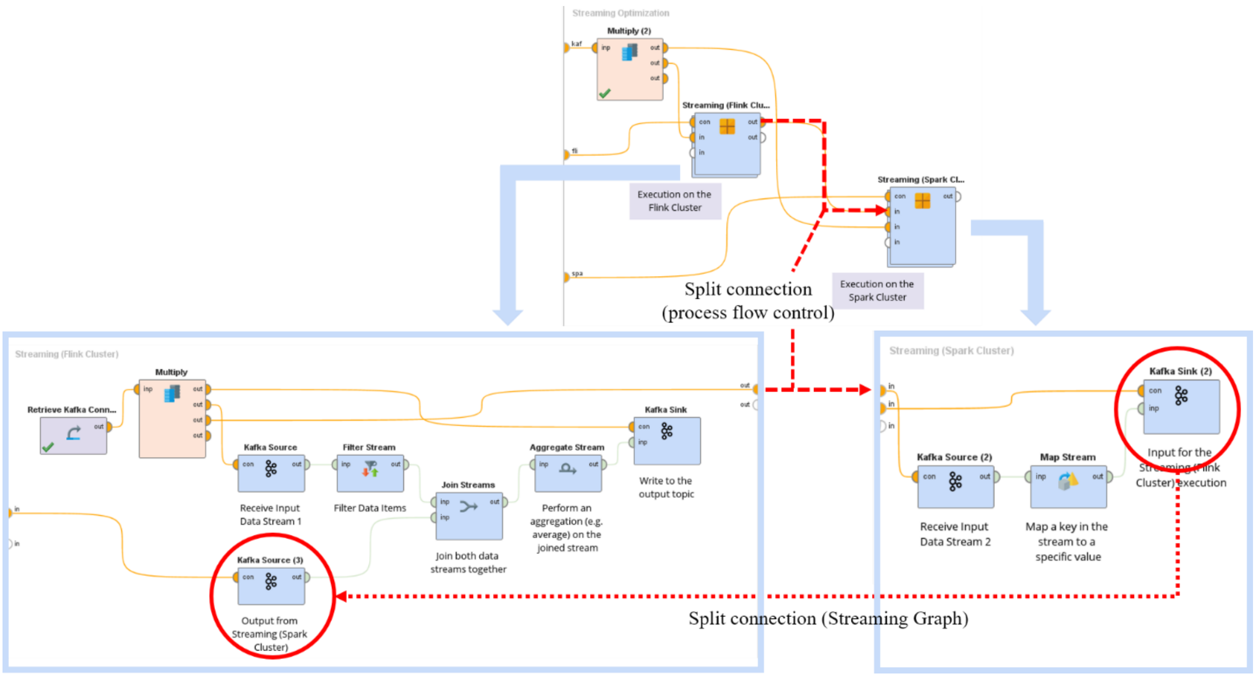

Streaming Optimization
Multiply (2)
kaf
inp
out
out
out
fli
Streaming (Flink Clu...
con
in
in
out
out
Execution on the Flink Cluster
Streaming (Spark Cl...
con
in
in
in
out
spa
Split connection (process flow control)
Execution on the Spark Cluster
Streaming (Flink Cluster)
Multiply
Retrieve Kafka Conn...
Kafka Source
Filter Stream
Join Streams
Aggregate Stream
Kafka Sink
Receive Input Data Stream 1
Filter Data Items
Join both data streams together
Perform an aggregation (e.g. average) on the joined stream
Write to the output topic
Kafka Source (3)
Output from Streaming (Spark Cluster)
Streaming (Spark Cluster)
Kafka Sink (2)
Input for the Streaming (Flink Cluster) execution
Kafka Source (2)
Map Stream
Receive Input Data Stream 2
Map a key in the stream to a specific value
Split connection (Streaming Graph)

Figure 17: Updated subprocess of the *Streaming Optimization* operator after the response of the Optimizer Service is received. Streaming Nest operators for the streaming execution backends are automatically created and the operator placed, according to the optimizer response. Split connections are re-established by either inserting *Kafka Sink* and *Kafka Source* operators (in case of a stream graph connection) or by using the *"in"* and *"out"* throughput ports of the *Streaming Nest* operators.

The optimized workflow is directly executed after it is updated. The different *Streaming Nest* operators deploy their inner workflows to the corresponding streaming execution backends. The actual optimization result can be visually investigated by checking the different *Streaming Nest* operators. They represent the placement of the operators according to the optimization.

With the *"Restore Process"* button, the original logical streaming analysis workflow can be restored by the user.

4.5 Integration of the Synopsis Data Engine Component

The interface and connection to Synopsis Data Engine (SDE) component is implemented via a RapidMiner operator with the same name. The operator has two input ports, one for the incoming data stream and one *"connection"* input port, which expects a *Kafka Connection* with the connection details to communicate with the SDE component. The SDE acts as a service component that can analyse an incoming data stream and return a synopsis of the data. For interacting with the SDE component additional nodes of the stream graph are required and created by the operator. The first is a SynopsisDataProducer that produces the data stream that goes into the SDE. The second is a SynopsisEstimateConsumer that receives the synopsis from SDE. Because the SDE exists as an independent service a third node is required, the SynopsisEstimateQuery, that periodically queries the SDE Service. Also, a SDERequest object is used to request the initial addition of selected synopsis in the SDE Service.

Figure 18 shows the different streams the operator creates internally. Upon execution of the process, an "*ADD*"-command to create and add the selected synopsis is written to the *request* Kafka topic (on the Kafka cluster defined by the *Kafka Connection* object provided to the operator) of the SDE Service (this is done by the SDERequest object. Then, the data items received from the data stream, which is connected to the input port, are written by the SynopsisDataProducer to the *input* Kafka topic of the SDE Service. Periodically, the SynopsisEstimateQuery node is writing estimate queries to the *Request* Kafka topic of the SDE Service to initiate an estimation of the synopsis. The SynopsisEstimateConsumer is consuming the *output* Kafka topic of the SDE Service. Data items written to this topic by the SDE Service are consumed, and then written to the output data stream, which is connected to the *"output*" port of the *SDE* operator.

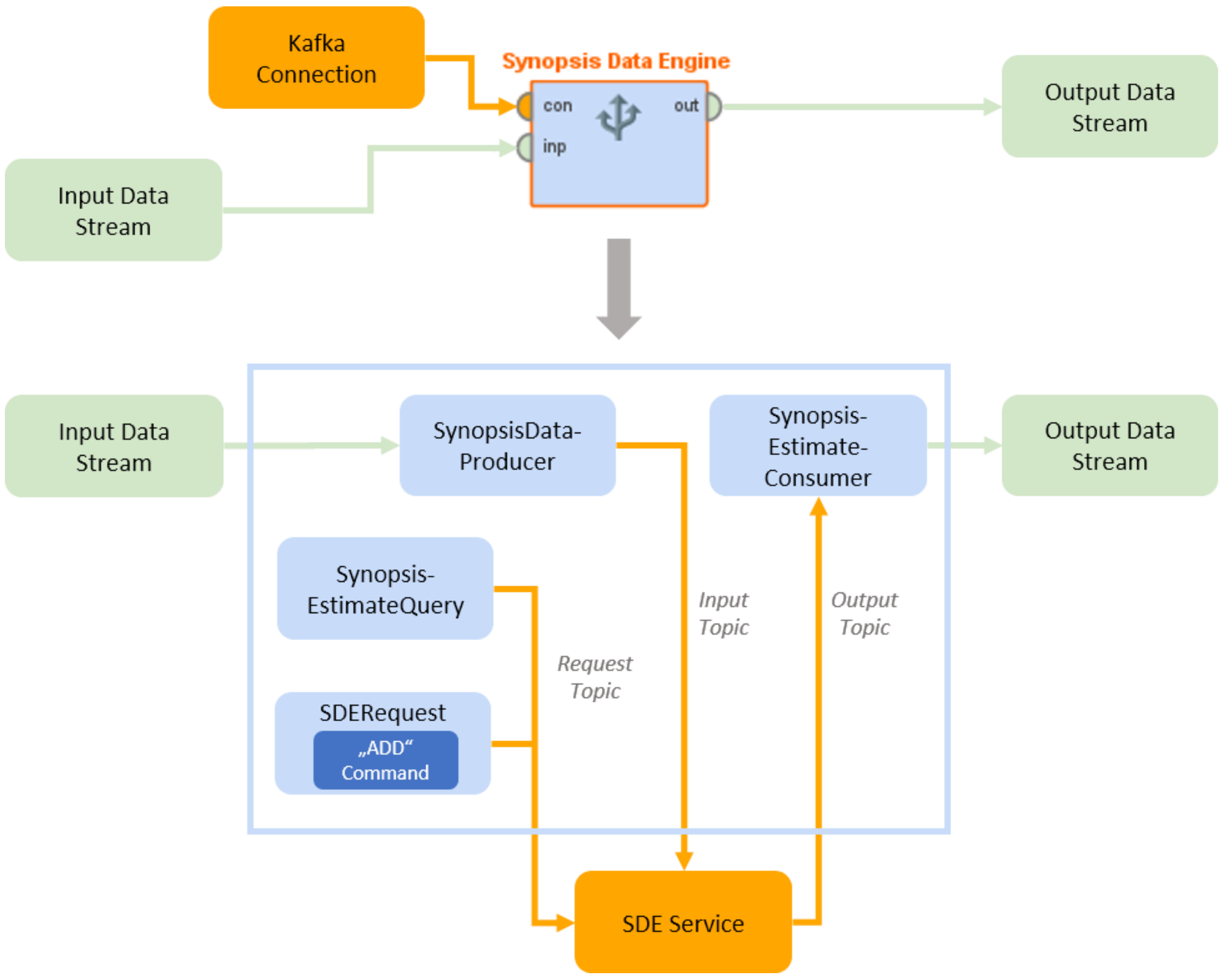


Figure 18: Overview over the different streams, which are created by the Synopsis Data Engine operator internally to communicate with the SDE Service.

Figure 8 shows a designed workflow using the new *Synopsis Data Engine* operator. Also, the parameters of the operator are shown. The parameter *"synopsis type"* defines the used synopsis. The SDE operator can support many different types of synopsises. For now, the *SDE* operator supports the following operations:

- Count min
- Bloom filter
- AMS
- DFT
- LSH
- Core Sets
- HyperLogLog
- Sticky sampling
- Lossy counting
- Chain sampler
- GK Quantiles

The parameters for the specific synopsis can be provided by the parameter *"synopsis params"*. The parallelism used in the computing of the synopsis is specified by the parameter *"synopsis parallelism".* The parameters *"data set key", "data value key, "u id"* und *"stream id"* are used to configure the corresponding ids and keys in the synopsis. The parameter *"estimate key"* specifies key of the estimated value (the value computed by the synopsis) in the output data stream. The frequency of queries for receiving the estimate by the SynopsisEstimateQuery can be defined by the parameter *"estimate frequency"*.

The parameters *"request topic", "data topic"* and *"output topic"* are used to configure the names of the Kafka topics used by the *SDE* operator to communicate with the Synopsis Data Engine Service.

4.6 Integration of the Interactive Online Machine Learning Component

The integration of the Online Machine Learning (OML) Component is achieved by an operator with the same name. The OML Service is deployed by the operator itself. The connection information to the Flink Cluster on which the OML Service shall be deployed (from now called OML-Flink-Cluster) must be provided to the *"flink-connection"* input port of the operator. The connection information to the Kafka Cluster used to communicate with the deployed OML Service (from now called Communication-Kafka-Cluster) must be provided as a *Kafka Connection* to the *"kafka-connection"* input port.

First the operator creates the Kafka streams to the necessary Kafka topics (*"training topic", "request topic", "forecast input topic", "forecast output topic"*) on the Communication-Kafka-Cluster. The user can specify the topic names by the corresponding parameters. Then the operator deploys the jar file of the OML Service (provided by the parameter *"job jar"*) to the OML-Flink-Cluster which starts the OML Service on the cluster. The Service is configured to use the earlier created Kafka topics, for communication.

An initial "Create" request is then written to the *request topic* to create and configure the algorithm used in the current streaming analysis workflow. The configuration includes the selected learner (configured by the parameters *"learner name"*, *"learner hyperparameters"* and *"learner parameters"*), the training configuration (configured by the parameter *"training config protocol"* and *"t*raining config m*ini batch size"*) and a list of preprocessor algorithms (configured by the parameter *"preprocessors"*, for each preprocessor the *"name", "hyperparameters"* and *"parameters*" must be specified).

The data items which are received from the data stream connected to the *"training input"* port, must have labelled information. They are written by the *OML* operator to the *training topic* of the OML Service. The data items, received from the data stream connected to the *"input stream"* port, are likewise written to the *forecast input topic* of the OML Service. This is achieved by two internally used KafkaSinks. An internal KafkaSource is used to consume the *forecast output topic* of the OML Service and provide the scored data items to the data stream connected to the *"output stream"* output port of the *OML* operator. Figure 19 visualizes the internal streams created by the operator and the different steps performed.

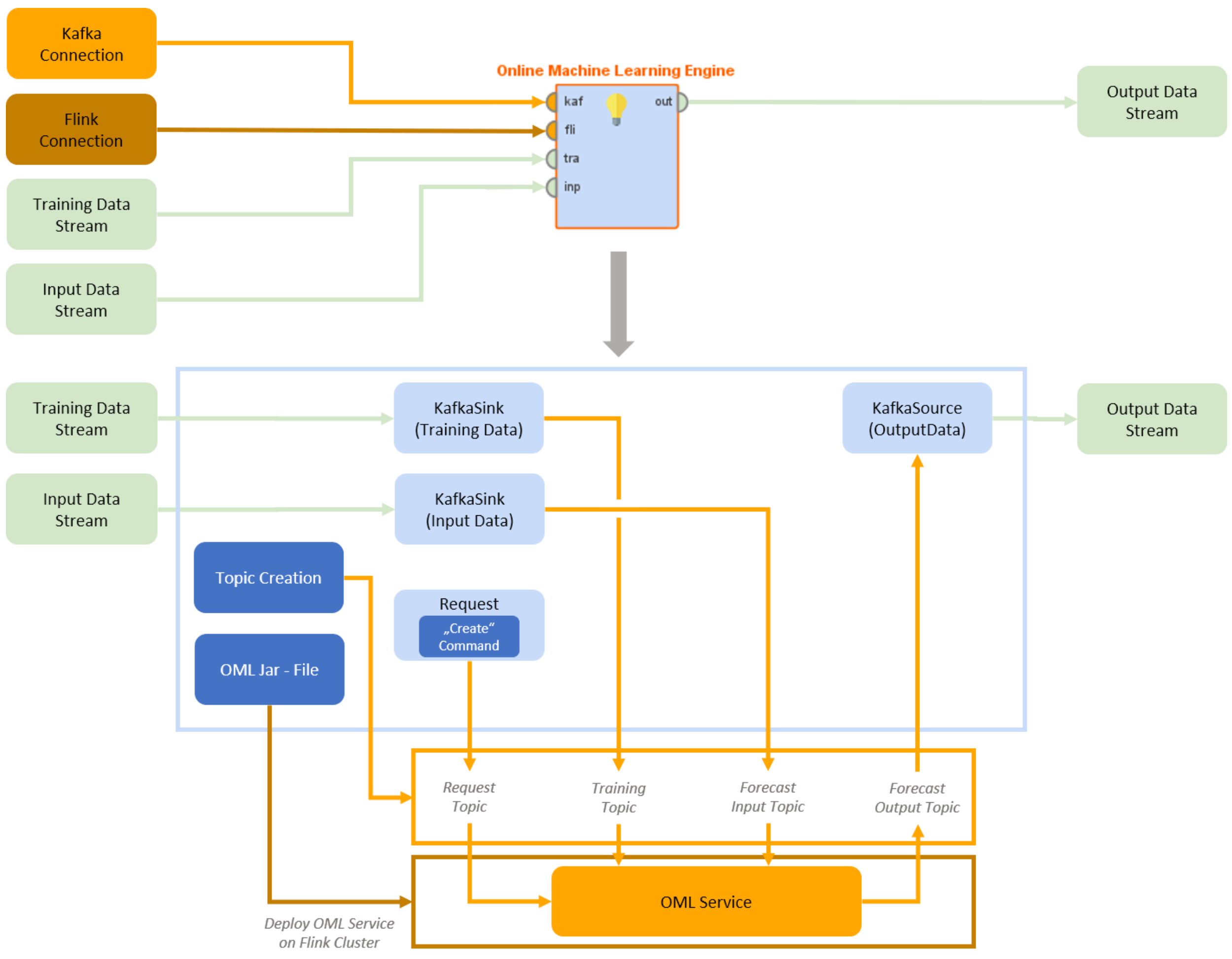


Figure 19: Overview over the internal structure of the *Online Machine Learning* operator. The diagram shows how the OML Service is deployed by the operator, as well as the internal streams created by the operator to communicate with the OML Service.

5 Analysis of INFORE Architecture

5.1 Advantages

The INFORE architecture targeted some of the most ambitious design, execution and management time requirements regarding streaming workflows. The various functional and non-functional aspects that were targeted as the aim of the project have met with fruition in the functional prototype that has been developed as a result of several iterations, critical evaluation of the state of art in streaming technologies, considering the use case requirements and keeping input from various project partners in sight. Based on the functional prototype already developed, we are able to clearly assess and report some of the advantages the INFORE architecture and system prototype is delivering at par and even beyond the state of art in creating, configuring, optimizing, deploying and managing complex streaming workflows on multiple streaming backends. Hence, we argue that INFORE technology can substantially contribute to enable the new breed of long running, high throughput and low latency streaming applications that utilize machine learning and smart data processing in complex industrial and scientific domains. Some of these advantages are concretely listed below.

- **Visual Approach to Create Streaming Workflows:** A *visual design approach* has been realized that eases creation of streaming workflows. To our knowledge, this is a severely under-addressed area in current art. A lack of visual tooling to create streaming workflows restricts many different personas such as non-programmers. INFORE has reused and extended the RapidMiner visual design (coding-optional) approach, which arguably has enabled a host of personas such as domain experts, scientists and analysts, who would otherwise be withheld due to the overwhelming burden to learn various streaming technologies and develop programming skills. INFORE's approach not only removes this burden by enabling to create streaming workflows visually, but these workflows also port to multiple backends.

- **Generic, Technology Agnostic and Multi-Platform Portable Design:** INFORE's functional prototype has successfully demonstrated its *generic design* for creating workflows that are *technology-agnostic* and *platform-independent* in terms of supported platforms. For example, INFORE workflows use transformations that have a logical-physical separation. This separation enables the user to focus on designing streaming workflows in general terms using logical operators, while the architecture translates these logical operators into their platform specific physical forms. Thus, INFORE approach allows us to write streaming workflows once, while executing them on any of its supported backends. This high level of abstraction delivers a flexibility that is beyond some of the popular streaming (abstraction) frameworks (e.g., Apache Beam), which address this generalization to some degree, but without a visual design approach or real-time optimizations that INFORE is capable of. The visual design approach coupled with platform-abstract execution alone makes INFORE solution highly plausible for adoption.

- **Workflow Optimization and Intelligent Placement:** Another *differentiator* for INFORE is its use of the Optimizer Component, which induces *real-time intelligence* in terms of re-configuring parameters of the workflow, its deployment and execution-time decisions. The INFORE Optimizer Component evaluates streaming workflows on basis of their functional (qualitative) and non-functional (quantitative) requirements and considers the infrastructural details of available streaming backends (network and resource information, etc.). The optimization determines the best placement strategy for a given workflow (to be used either as a singular whole or split into multiple workflows) as well the exact backend(s) for executing the workflow(s). Moreover, the optimized workflow(s) can be rendered in the visual Graphical Editor component (RapidMiner Studio GUI), so that the human user (workflow designer) can inspect the results of optimization and reconfigure the Optimizer (if need be).

- **Futureproofing Workflows through Extensibility and Pluggability:** INFORE is designed for *extensibility* and *pluggability* from its foundations upwards, with the intent to keep the streaming workflows as *future-proof* as can be envisioned at this stage. The extensibility of the architecture is achieved (to a large degree) by the Manager Component, which provides programmable modules, using which, new features can be added to the INFORE prototype in a pluggable manner. Following this inclusive, extensible and pluggable approach, new operators can be added rather conveniently to the Streaming Extension. This enables not just the project partners, but also third-party developers and potentially all following adopters, to implement operators for new sources, sinks, flow-control, transformations and/or integration operators (to interface with custom components or services) – using

INFORE’s standardized programmable approach. Hence, the outcome of INFORE, or any of its successors, is expected to have an impact beyond the scope of INFORE as a project.

- **A Reliable, Robust and Flexible Middleware for Interfacing Disparate Components:** The INFORE architecture had to integrate some highly complex components that offer very specialized and advanced features, such as online machine learning and generating continuous synopsis on streaming data. Many of these components posed integration challenges due to various reasons, including but not limited to distribution and scaling requirements, latency guarantees, configuration and computational requirements, upstream, downstream and lateral dependencies. These challenges have been resolved in INFORE by a broad adoption of Kafka as the connection middleware to achieve integration among disparate services – all of which can be seamlessly accessed, in a location-transparent manner, using the integration operators developed in the Streaming extension. Leveraging Kafka to achieve this backbone-level interfacing on one hand, allowed INFORE streaming workflows to benefit from the novel facilities provided by SDE and OML components, but on the other hand, also led to the realization of a highly reliable, robust, flexible and common interfacing approach to integrate disparate components and services not just limited to those listed here.

5.2 Release of the INFORE Software Stack and Extensibility of the INFORE Architecture

The individual implementations of the INFORE components will be publicly available at the project bitbucket repository (https://bitbucket.org/infore_research_project/). The different components are implemented by the following repositories:

- **Connection Component, Graphical Editor Component, Manager Component:**
  RapidMiner Studio Streaming Extension: https://bitbucket.org/infore_research_project/rapidminer-extension-streaming

- **Optimizer Component:**
  Workflow Optimizer: https://bitbucket.org/infore_research_project/optimizer
  Metric collection tools: https://bitbucket.org/infore_research_project/platformstatistics

- **Synopsis Data Engine Component:**
  Synopsis Data Engine (SDE): https://bitbucket.org/infore_research_project/6.1-sde
  A Client for the Synopsis Data Engine (CSDE): https://bitbucket.org/infore_research_project/a-client-for-the-synopsis-data-engine-csde

- **Online Machine Learning Component:**
  Online_Machine_Learning_via_Flink:
  https://bitbucket.org/infore_research_project/online_machine_learning_via_flink
  Online Machine Learning Clients:
  https://bitbucket.org/infore_research_project/online_machine_learning_client

The graphical workflow design component is available as RapidMiner extension via the code repository of the project. The compiled extension can be added to any existing RapidMiner installation and allows the user to create and submit their own graphical workflows to the supported streaming platforms (Apache Flink, Apache Spark). Furthermore, the RapidMiner API provides the option to write own operators based on the INFORE architecture. With this possibility developers can provide code free realization for specific operations for the platforms, which are not yet provided by the INFORE prototype. It’s also possible to use the architecture as a blueprint for developing the support of more platforms and streaming engines. This is the same approach the INFORE project team will follow in further iterations to enhance the architecture.

The full guide on how to use the RapidMiner developer API for writing extensions is available here: https://docs.rapidminer.com/latest/developers/extensions/

5.3 Graphical Specification of complex Data Processing Workflows with the Prototype of the INFORE Architecture

This section showcases how the graphical specification of complex data processing workflows with the initial prototype of the INFORE Architecture, which is described in this deliverable, can be performed. The example workflows shown in the following subsection illustrate how workflows for the Life Science Use Case (see work package 1) and the Financial Use Case (see work package 2) could look like. For this initial state of the prototype and the evaluation of the use cases in the INFORE project, these are only demonstration workflows. Some of the operators shown are only in the concept phase and are not yet implemented. In the future development, the functionality of the INFORE Architecture will be enhanced to support the analytic needs of the use cases in the INFORE project. The designing and implementation of the workflows for the different Use Cases will therefore also function as a good evaluation of the usability of the INFORE Architecture.

5.3.1 Complex Data Processing Workflow for the Life Science Use Case

Figure 20 shows an example workflow for the Life Science Use Case. It is only a demonstration of a how a streaming analysis workflow could look like for this use case. As the execution of the workflow shall be optimized, the streaming part of the workflow is designed inside the subprocess of the *Streaming Optimization* operator. As described in Section 3.2.4 and 4.4 a *Kafka Connection* for the inter communication between different computing clusters is provided to the *Streaming Optimization* operator. Also, a collection of *Flink Connection* and *Spark Connection*, which are the execution sides for this example workflow, are provided to the *Streaming Optimization* operator.

In the designed workflow, labelled Simulation Data is received by the *Kafka Source (2)* operator and provided to the *Synopsis Data Engine* operator, which computes a discrete Fourier Transformation representation of the input data. The output is provided to the *Online Machine Learning* operator which trains an online Support Vector Machine (SVM) model on the input data. The target variable of this model is the usability (in terms useful simulation results, see deliverable D1.1 for a description of the Use Case scenario). In addition to the model, the *Online Machine Learning* operator produced a test results set, which is used by the *Validate Model* operator to evaluate the performance of the trained online SVM. The model is then provided to the *Apply Model* operator.

Unlabelled Simulation Data (from currently running simulations) is retrieved by the *Kafka Source* operator and provided to the *Apply Model* operator. This operator uses the model to tag the unlabelled Simulation Date and predict the usability of the running simulation.

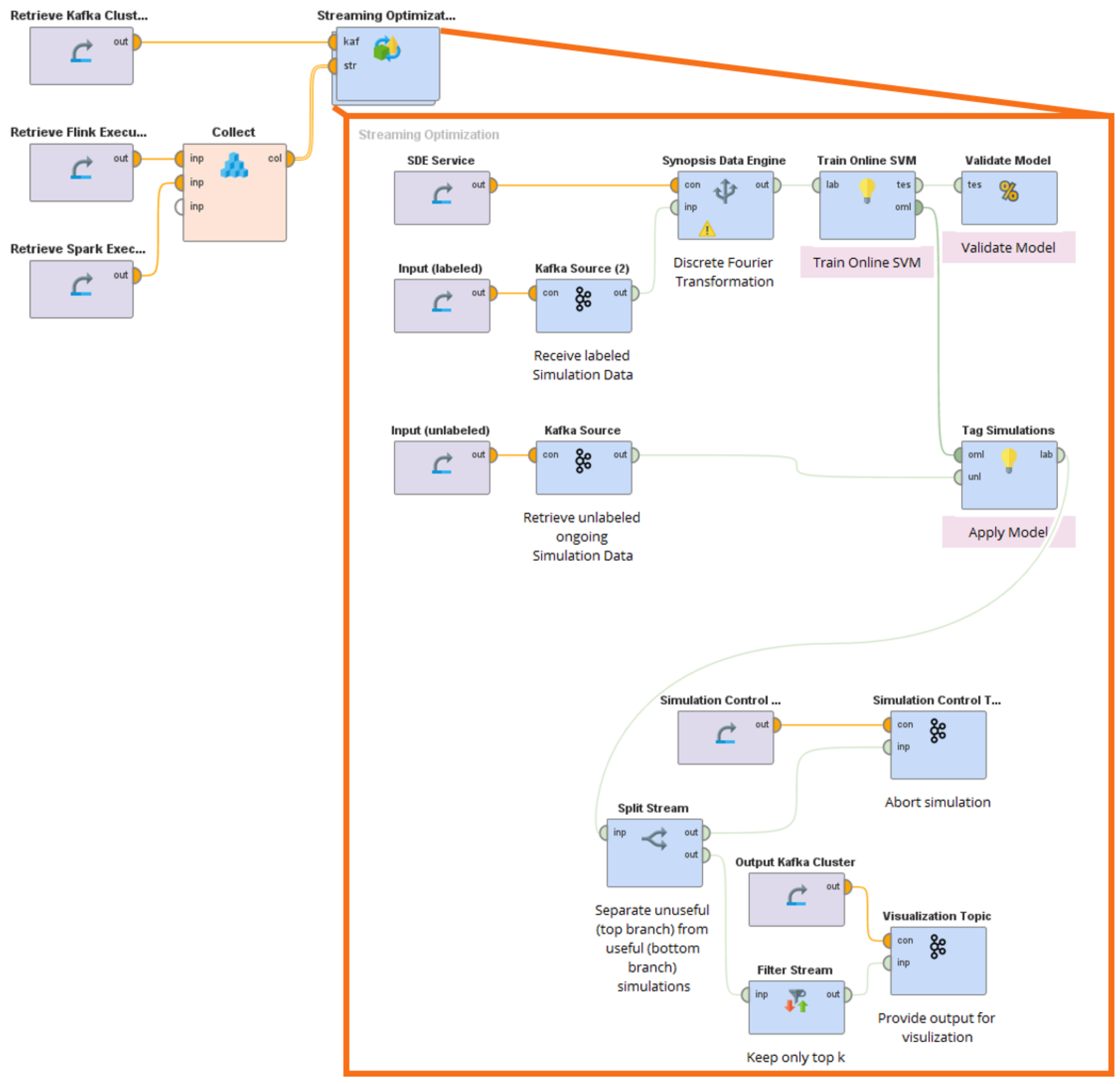


Figure 20: Complex data processing workflow for the Life Science Use Case. The target of this streaming analysis is to train an online machine learning model able to predict the usability of running simulations. This is then used to abort non useful simulations and visualize the useful ones.

The *Split Stream* operator receives the tagged Simulation Data and split it between non useful simulations (top branch) and useful simulations (bottom branch). The *Simulation Control Topic* operator (which is a *Kafka Sink* operator) sends "abort simulation" commands to the *Simulation Control Kafka Cluster* to abort the running non useful simulations.

The useful simulations are filtered by the *Filter Stream* operator, so that only to top k simulations are provided to the *Visualization Topic* operator (which is a *Kafka Sink*) operator. This top k simulations are written to the output Kafka topic which is used to visualize the data.

5.3.2 Complex Data Processing Workflow for the Financial Use Case

Figure 21 shows an example workflow for the Life Science Use Case. It is only a demonstration of a how a streaming analysis workflow could look like for this use case.

Like the example workflow for the Life Science Use Case (see Figure 20), the streaming analysis workflow is designed in the subprocess of the *Streaming Optimization* operator, so that the execution of the workflow is optimized. The corresponding connection objects needed by the *Streaming Optimization* operator are provided to the input ports of the operator (see Section 5.3.1 for a description).

The *Input Source* (which is a *Kafka Source* operator) retrieves the financial input data. The *Split Stream* operator splits the input data into Level 1 (transactions, top branch) and Level 2 (offers, bottom branch) data streams. See deliverable D2.1 for a more detailed description of the use case scenario definition and the meaning of Level 1 and Level 2 data.

Both streams are filtered with corresponding *Filter Stream* operators to filter on subsets of individual stocks. The *SDE* operator holds a *count min* synopsis of the Level 2 data, while the *Project Data* operator projects the Level 1 data to only contain the actual transaction timestamp. The *Join* operator joins the two streams together, so that the output stream has a count min of the Level 2 data (hence the count min of the offers) for each transaction in the Level 1 data. This count min is a good approximation of the "activity" for the corresponding stock.

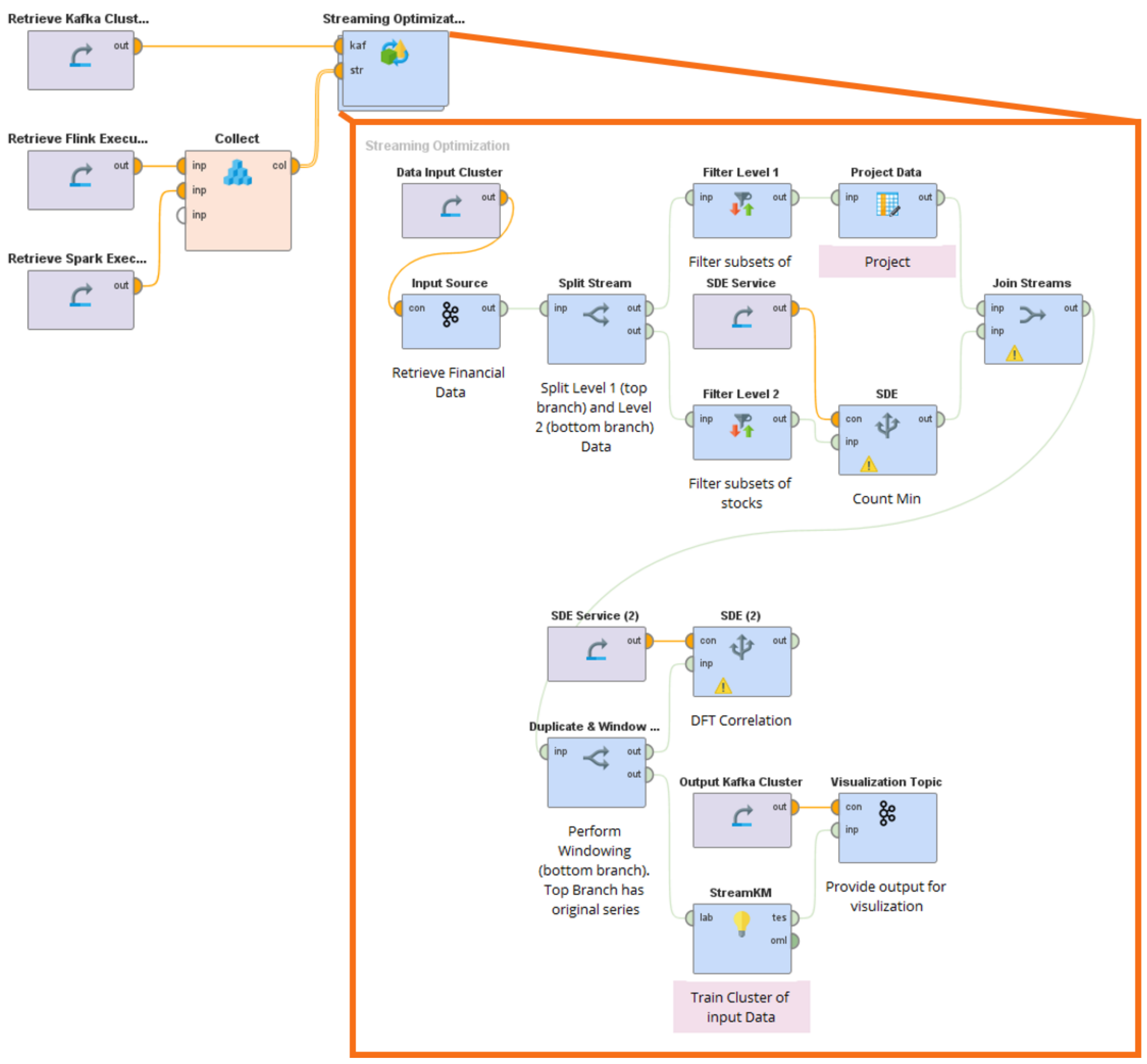


Figure 21: Complex data processing workflow for the financial use case. The target of this use case is to create an approximation of the "activity" in stocks, which is used to identify correlations and clusters in the stock market activities.

This output data is further processed by the *Duplicate & Window Stream* operator which provides the original stream to the top branch and a windowed stream to the bottom branch. The *SDE (2)* operator computes the correlations on the original input stream using Discrete Fourier Transformation. The windowed stream is used to train a cluster model by the *StreamKM* operator (which is an *Online Machine Learning* operator). The results of the cluster model are provided to the *Visualization Topic* operator (which is a *Kafka Sink* operator), which is used to provide the output data for visualization.

6 Conclusion

The developed software stack presented in this deliverable is a functional prototype for the INFORE Architecture designed and presented in the deliverable D4.1. The interaction and integrations of the different components and the developed implementations allow us to accomplish the main objectives of the INFORE project. It supports the non-programmatic approach for designing and deploying streaming analysis workflow. The platform independent abstraction layer allows that the user of the INFORE Architecture does not need to bother about technological details of streaming analysis functionality and allows as well for the integration of the optimization of the designed analysis workflow, thus, leveraging the possibility to tackle extreme-scale analytic use cases in an optimized fashion. The integration of the Synopsis Data Engine and the Online Machine Learning components allows for leveraging the functionality of real-time, interactive machine learning and data mining, data summarization and approximate query processing techniques in these use cases.

The prototype is developed in a holistic, pluggable, extensible approach, to enables further improvements of its functionality. The development of the next months will focus on improving existing functionality and evaluating the usage of the INFORE Architecture, by implementing workflows of the Use Case partners in the INFORE research project. By collecting feedback for testing and usability from the different stakeholders we can enhance this initial prototype of the INFORE Architecture to a complete prototype of the designed Architecture, able to fully support all targets of the INFORE project. The final system prototype is planned to be completed in Month 32 of the project and will be documented in the Deliverable D4.3.

7 References

[1] Pankaj Misra, Tomcy John. “Data Lake for Enterprises”. Packt Publishing. 2017. ISBN: 9781787281349

[2] J. Kreps, N. Narkhede, J. Rao. “Kafka: a distributed messaging system for log processing”. The 6th International Workshop on Networking Meets Databases, Athens, Greece, 2011.

8 Appendix

8.1 Section 4 of Deliverable D4.1: Definition of the INFORE Architecture

This section is from the deliverable D4.1 and describes the definition of the INFORE Architecture, as defined in the D4.1 deliverable. As the initial prototype, described in this deliverable D4.2, is the implementation of the Architecture, this section is added for convenience.
This section describes the structure and concepts defined for the architecture and how the different components interact with each other.

8.1.1 User Requirement Analysis

While the initial concept and requirements of the architecture were already envisioned during the project proposal phase, some refinements are always required during the starting phase of the project and while the system prototype takes shape. To ensure that all necessary requirements of the use case partners are met, an initial phase of requirement engineering was initiated in parallel with the requirements expressed in deliverables D1.1, D2.1 and D3.1.

Here, the concept of a target persona and user stories were applied. These concepts are typically used by software development projects and designers. A persona represents the archetypical users of a tool and helps us as developers to better identify them. It consists of a sketch about the daily work of that person, her educational background, often used tools and some common personality traits to flesh out the characterization. The user story is another element that helps to refine the needs of the user. It is a simple sentence that states the role of the user, her desired action and the goal that she wants to achieve. Each user story represents one task that is crucial for the work routine of a persona.

We asked the three use case partners from WP1, WP2 and WP3 to create sample personas of the latter users of the developed tools and the typical tasks they want to perform.

From all partners, we have collected in total seven different target personas that represent typical users of the final system. Additionally, six user stories where formulated that help to define the tasks these users would later like to perform. While these requirements focus more on the end users and their experience with the final tool, they already helped to formulate requirements, especially on the user interface and the graphical editor.
It became clear that the requirements are not perfectly aligned for all partners. For some user types the multi-platform approach is crucial, as they need a system that eases the integration of different data streams and analysis platforms. For others (e.g., in the financial use case), the high level of interactivity and easy representation of data streams is more important. These findings again stressed out the importance of a flexible architecture that can handle the heterogeneous requirements for streaming applications.

8.1.2 Architecture Overview

The INFORE Architecture is designed to tackle the project objectives described in section 3.1 (of deliverable D4.1). Conceptual pillars are used to create a component-based modular design. Figure 22 gives an overview of the conceptual design of INFORE Architecture. The integration concept is described in the next subsection, while the different components are explained in detail in the following subsections.

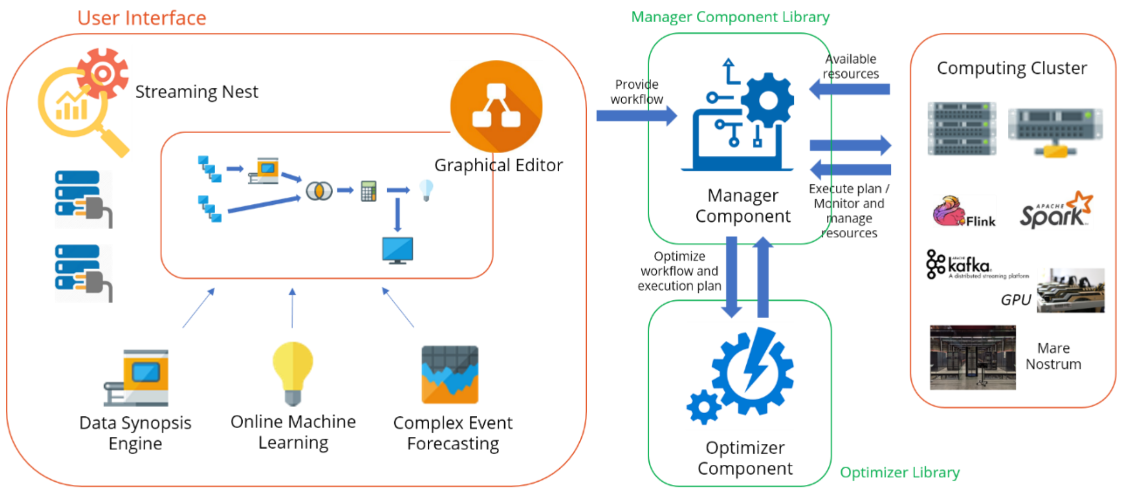


Figure 22: Overview of the conceptual design of the INFORE Architecture.

8.1.3 Integration

The INFORE Architecture will achieve design time and execution time reduction goals through a holistic integration approach based on *Conceptual Pillars*. These pillars are concretely realized as components which are the following:

1. Connection Component
2. Graphical Editor Component
3. Manager Component
4. Optimizer Component
5. Synopsis Data Engine Component
6. Complex Event Forecasting Component
7. Interactive Online Machine Learning Component

The central component is the Manager Component providing the interaction point between most of the other components. The Manager Component handles the actual execution and monitoring of the designed streaming analysis process.

All components will be developed in a modular way, with clear interaction-interfaces between them. This enables an easy exchange between the modules and provides the necessary pluggability of the INFORE Architecture.

8.1.3.1 Connection Component

The Connection Component enables access and preliminary management of the selected streaming backends (stream sources and sinks, Big Data platforms and respective computer clusters) that are utilized by the INFORE use cases and broader application scenarios. The Graphical Editor Component will be a streaming extension of the RapidMiner Studio (which is currently focused on offline, batch processing) to provide specific functions (as operators) that are developed in the selected streaming technologies.

Some aspects of streaming technologies being considered at this stage are explained in Section 8.1.2. Here, we briefly mention the mechanics of the Connection Component. The Connection Component functionality is realized as Connection objects in the Graphical Editor Component. The user of the INFORE Architecture only needs to provide the essential information for connecting to the backends without any coding needed. Then, the Connection objects can be utilized in the streaming analysis process, which itself is designed using the Graphical Editor Component.

For the scope of the INFORE project, we identified three functional concepts that include input streams, output streams and stream processing backends, where a streaming analysis process may be executed. A typical streaming analysis process will consume one or more input streams, perform transformations and may result in output stream(s). Input streams provide the input data, for which the streaming analysis process is designed. The streaming analysis process is application specific. Output data streams (which are not a necessity) mark the endpoint of a streaming analysis process. The nature of the application scenario can impose restrictions on their type, but the approach provides the required flexibility. Output streams can serve as Input streams for downstream analysis processes. Streaming processing backends are the computer clusters typically hosting some Big Data platform(s), which provide the physical infrastructure resources needed to execute the streaming analysis process. These are limited by the hardware and technology stacks the user has access to.

Keeping the above description as a pretext, the connectivity requirements become very important. The Connection Objects are thus configured for each supported compute backend e.g., to connect with Flink or Kafka backends. The Connection Object(s) are then used together with the Manager Component to establish connection for input stream, output stream, or to place a streaming analysis process on a compute backend and/or to configure environmental settings. The Connection Component can be extended with additional Connection objects to reflect the execution requirements of the streaming analysis process. For instance, if a streaming backend technology provides a set of desired connectors, which are installed in the Architecture implementation, then, these may be reused by the streaming analysis processes. As an example, a Flink-based streaming analysis process may output its results into a Kafka topic. In such a case, INFORE's Flink Connection object can encapsulate the Flink connectors for required sources (for input streams) and sinks (for output streams) to provide access in real time. A user can graphically define and internally create multiple Connection objects based on the available backends.

The concepts of input streams, output streams and stream processing backends capture the general constituents of a typical streaming analysis process. With some exceptions, all streaming technologies described in Section 8.1.2 can provide the actual functionality for these constituents, which are addressed in Section 3. For flexibility, the INFORE Architecture aims at combining different streaming technologies in one streaming analysis process. If more than one stream processing backends are provided to the INFORE Architecture, the architecture by virtue of the Optimizer Component can automatically decide on the optimal placement of different parts of the analysis process to different backends (Big Data platforms and/or cluster/HPC infrastructures).

8.1.3.2 Graphical Editor Component

The Graphical Editor Component enables users of the INFORE Architecture to easily design a streaming analysis workflow without any coding needed. This is achieved by encapsulating streaming analysis functionality into so called operators. The operators can be placed inside the graphical user interface via drag and drop actions and can be connected by drawing arrows to define the data flow in the streaming analysis workflow.

To ensure flexibility, the streaming analysis operators of the Graphical Editor Component, called "Logical Operators", are designed as an abstraction layer, without defining the actual streaming technology used. Thus, the user can focus on the analytic setup of the process, without handling the technology specific aspects.

The Manager Component can automatically perform the desired functionality by utilizing the physical implementation(s) of the Logical Operators for the specific streaming processing backend used. If physical implementations for a particular Logical Operator in more than one streaming processing backends, the Optimizer Component can devise the selection of the physical implementation to optimize the processing of the workflow. This is of course only possible for functionality (operators) for which multiple implementations over different Big Data platforms exist. See Section 3.1 (of deliverable D4.1) for an overview of the common streaming technologies.

The functionality of the Data Synopsis Engine Component, the Complex Event Forecast Component and the Interactive Online Machine Learning Component are represented as Logical Operators as well and can be placed (dragged and dropped) in the streaming analysis workflow and get graphically parameterized. The physical implementation of these operators on top of specific Big Data platforms (run on respective clusters) is developed within the scope of WP6.

These "INFORE Component" operators contain all information to execute the specific algorithms and methods, which are provided by external (external from the Graphical Editor Component's perspective) libraries. They provide loose coupling between the Graphical Editor Component and the specific INFORE Component. The libraries can be developed independent of the Graphical Editor Component. An adaption is only needed if the interface changes. In addition, further components can be added in the future by just adding operators to the Graphical Editor Components. These aspects ensure the extensibility and pluggability of the INFORE Architecture. The workflow is to be fed to the Manager Component, which also handles the deployment and execution aspects on the streaming cluster.

Figure 23 shows a demonstration of a streaming analysis workflow, designed with the Graphical Editor Component.

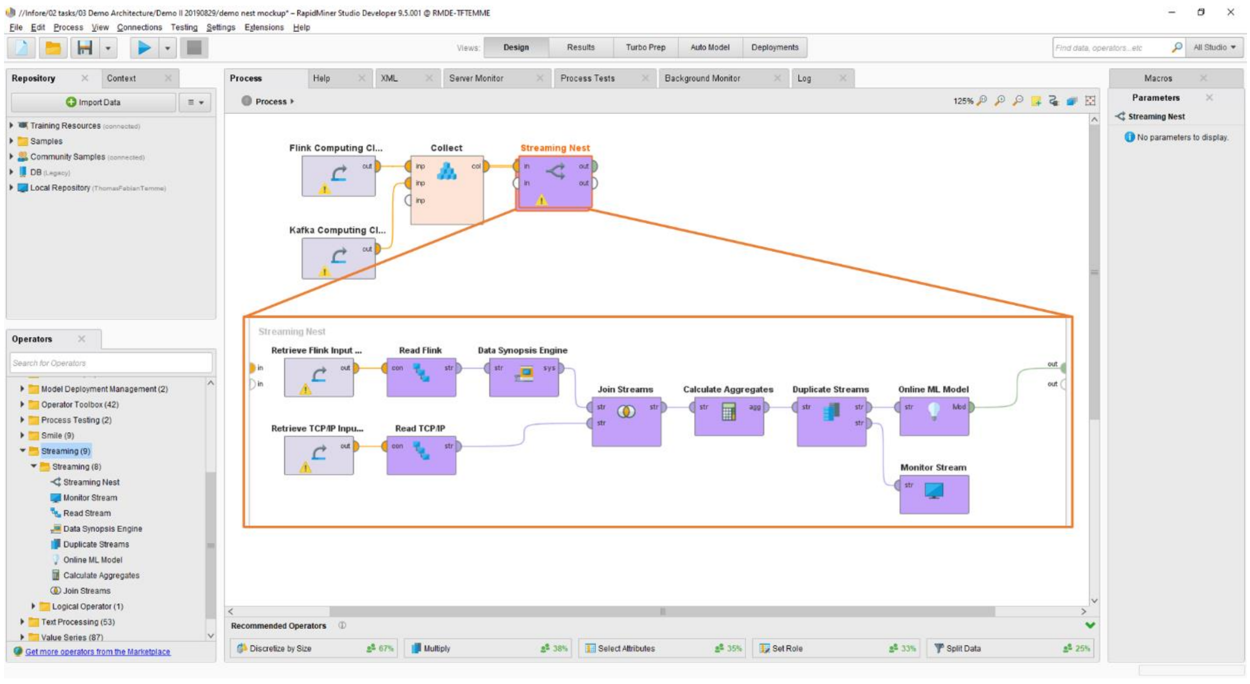


Figure 23: Demonstration of a streaming analysis workflow designed with the Graphical Editor Component.

8.1.3.3 Manager Component

The Manager Component plays a central role in the INFORE architecture. The Manager Component is realized as a library, which allows extensibility and flexibility to integrate with potentially multiple systems or front ends. This arguably sustains the work done in INFORE beyond the scope of the project. On one hand, the Manager Component wraps the *data retrieval*, *preparation*, *transformation* and *modelling* methods that are interfaced with the Graphical Editor Component in a loosely coupled fashion. On the other hand, it utilizes the Connection Component to handle connectivity with the streaming backend(s), provides a tool-agnostic representation of the streaming analysis workflow (as required by the Optimizer Component to optimize the workflow), and handles the deployment aspects. The Manager Component is also intended to retrieve real time information from stream processing backends on the available capacity, usage of resources, workload status, and stream processing meta data.

There are three functional modalities which lie in the domain of the Manager Component. These are briefly explained below.

Workflow and Plan related functions:

The Manager Component receives the workflow from the Graphical Editor Component. Besides the actual streaming analysis process, the workflow also contains the connection objects for the input and output streams and the computing backends, on which the workflow shall be deployed and executed. The Manager Component converts the workflow, the resource and execution information into a JSON representation. This representation acts as a tool-agnostic workflow representation, which the Optimizer Component can optimize in a domain and technology independent fashion.

After performing the optimization of the workflow, the Optimizer Component passes it back to the Manager Component. The next steps involve the deployment of the optimized workflow or parts of this workflow on the desired backend(s) (Big Data platforms and available clusters). The basic idea is that the Manager Component can deploy an optimized workflow, or even an unoptimized (default) workflow. The latter may be the case if optimization is not possible, for instance, due to lack of operator physical implementations in more than one Big Data platforms.

Deployment related functions:

The Manager Component also implements the functionality to create a deployable artefact from the optimized workflow and dispatch it. Recall that the workflow or parts of the workflow may need to be deployed at different backends. The deployable artefacts are simply referred here as a plan to differentiate the streaming analysis workflow from its deployable unit(s) e.g. a Flink job or a Kafka program. The deployment functions provide the fine link, where the generic design of INFORE streaming analysis processes meets with the underlying execution technology.

Infrastructure related functions:

The Manager Component also provides basic monitoring and control functions. These help to understand and manage the infrastructure context. Monitoring is needed on two levels. First, the execution status of the workflow may need to be queried at some intervals, and secondly, the usage of resources and current execution workload may need to be fetched by the Optimizer to assess whether the desired KPIs are being met.

This process-level and system-level feedback assists the Optimizer to update its variables related to the execution time decisions it makes. This may result in an adaption of the deployment plan. The control functions wrap the basic functionality to scale the resources up or down. The idea is to wrap the API or commands provided by the streaming technology stacks, to easily fetch or execute basic monitoring or control functions but not over-emphasizing on standardization since many of these features are frequently updated and work with specific versions of third-party utilities. Hence, these dependencies need to be installed on the compute backends as well.

The Manager Component is also able to update the visual representation of the workflow in the Graphical Editor Component according to the optimized workflow, retrieved by the Optimizer Component.

Figure 24 illustrates the functionality of the Manager Component.

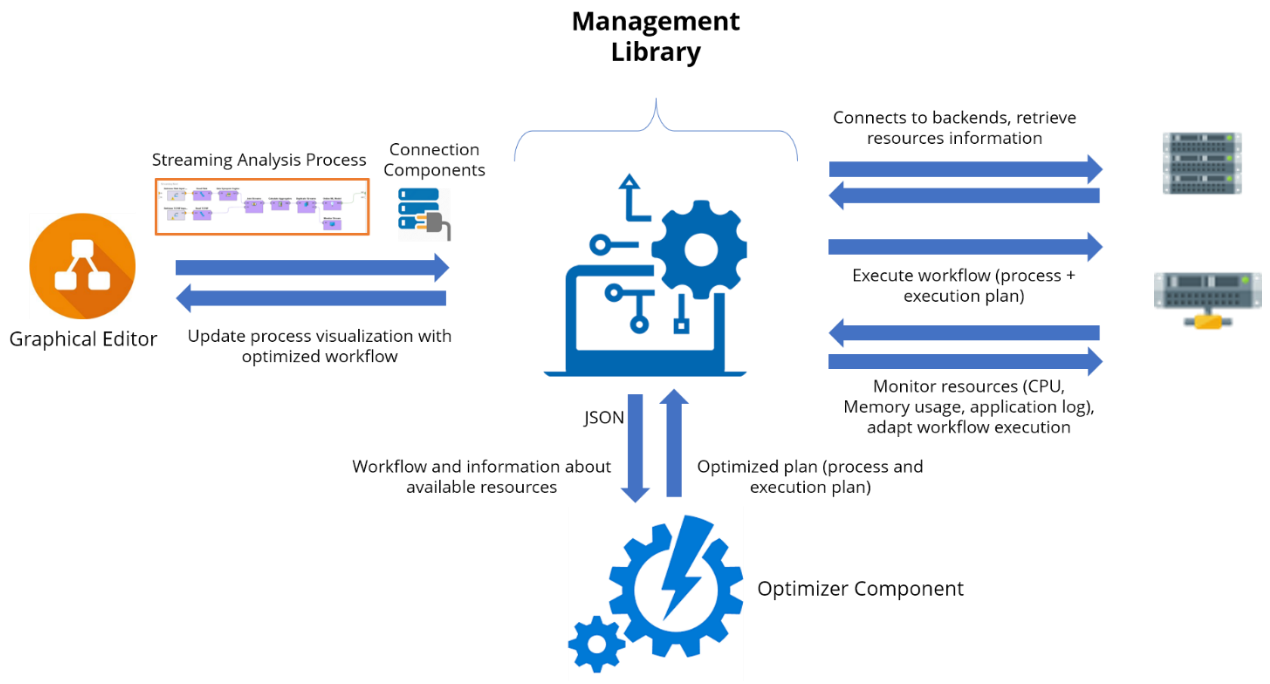

Management Library
Streaming Analysis Process
Connection Components
Graphical Editor
Update process visualization with optimized workflow
Connects to backends, retrieve resources information
Execute workflow (process + execution plan)
Monitor resources (CPU, Memory usage, application log), adapt workflow execution
JSON
Workflow and information about available resources
Optimized plan (process and execution plan)
Optimizer Component

Figure 24: Overview of the functionality which is provided by the Manager Component. The diagram clearly shows the central role which the Manager Component takes in the INFORE Architecture.

8.1.3.4 Optimizer Component

The INFORE architecture aims at processing workflows involving large-scale streaming data. These are complex workflows, typically spanning multiple execution and storage engines. Identifying an efficient workflow execution could involve decisions made on more than one execution engine. For example, consider a stream involving two engines, Kafka and Spark. It is possible to optimize the stream on each engine using best practices for Kafka and Spark, respectively. Furthermore, engines employing an intrinsic optimizer (e.g., Spark Catalyst) can provide an efficient execution plan for the part of the workflow running on the said engine. However, no engine has a complete picture of the workflow.

INFORE Optimizer helps with providing a holistic approach covering the entire workflow. Note, that INFORE Optimizer does not aim at replacing the engine specific optimizers. Rather, it works complementary by identifying optimization opportunities outside an engine and enabling further intra-engine optimizations by actions like function shipping (i.e., move a computation closer to the data) and data shipping (i.e., move the data closer to the computation). A basic optimization for the example Kafka-Spark workflow would be to push a filter from Spark down to Kafka to reduce the amount of data shipped over to Spark.

Workflow optimization in INFORE is based on a multitude of optimization objectives and configuration or system parameters such as resource availability, resource efficiency, workload parameters at runtime, efficiency of streaming technologies, availability of operator implementation on multiple engine, availability of execution engines, and so on. Typical goals considered by the optimizer include increase resource utilization, reduce latency, improve quality, satisfy business and technical constraints. The multi-platform approach of the optimizer allows us to efficiently handle and select the best suited resource available. For example, a join operation on multiple streams is a common implementation on most frameworks and so the optimizer can decide which platform to use, for example based on network locality or on available compute resources.

The Optimizer receives a workflow from the Manager Component. Then it optimizes the workflow (if applicable) and then sends the optimized workflow back to the Manager which handles workflow execution. This operation can be done either offline or online, during the running execution of a workflow, which allows the INFORE Architecture to adaptively react to changing condition of data streams and available resources.

The workflow metadata describe the designed streaming analysis process, the available resource information, the involved engines from the connected streaming backends, and user design choices. This information is encoded in a JSON format. An informed design choice we made is to decouple the workflow encoding of the Manager Component from that of the Optimizer. This is based on two reasons. First, in doing so, both the Manager and the Optimizer Components can be replaced by other tools if needed. This allows for increased pluggability and easy enhancement of the INFORE Architecture, and of potential future reuse of the code in follow-up applications.

The second reason was to enable an engine-agnostic workflow representation inside the Optimizer. Hence, a workflow designed for specific engines (e.g., Spark) is engine specific. For example, consider a workflow getting data directly from Kafka and containing a filter operator and a join operator implemented in SparkSQL. When the workflow is propagated into the Optimizer, it is converted to an engine-agnostic form that contains a logical filter operator and a logical join operator. This enables the Optimizer to look for optimization opportunities in other available engines (e.g., Kafka). A possible scenario for the example Kafka-Spark workflow could be as follows: (a) first, the workflow is transformed to an engine agnostic form and thus, the Spark filter and join are converted to an engine-agnostic filter and join, respectively; (b) then, the workflow is processed by the Optimizer that may identify an opportunity to push the filter back to Kafka; (c) next, the Optimizer converts the workflow to an engine specific workflow having two parts, one part with a filter to be applied to Kafka and one part with a join to be executed in Spark.

Besides workflow parsing, the Optimizer includes a component for statistics collection that keeps a history of statistic observations over past workflow execution, at the workflow level, at the operator level, and at the engine level. Once a workflow is sent to the Optimizer, the Optimizer enumerates the space of possible and promising execution plans for the workflow and estimates plan costs using a dynamic cost model that predicts workflow execution runtime based on the historical statistics collected. The navigation of the execution plan space can be done exhaustively or in a greedy fashion using heuristics for improved performance.

The optimization methods are described in more detail in the deliverables D5.1, D5.2 and D5.3.

8.1.3.5 Synopsis Data Engine Component

The Synopsis Data Engine (SDE) Component provides implementation of approximate query processing algorithms that can generate representative summaries on top of streamed data. The component includes a library implementing the query processing algorithms and operators, containing the configurations and parameters necessary to perform such a query to the SDE.

The usefulness of the SDE within INFORE streaming workflows is as follows. First, it can provide various kinds of scalability, including:

- Horizontal Scalability: although Big Data platforms such as Storm, Flink or Spark are destined by design to scale out the computation by parallelizing the processing load to a number of available processing units, the SDE can further boost horizontal scalability by working on carefully-crafted summaries of data and finally provide estimations of an operator's result, with accuracy guarantees. In that the processing load is shed due to the use of synopses and the computational complexity of the problem at hand is reduced.
- Vertical Scalability: this type of scalability concerns scaling the computation with the number of processed streams. Synopses provided by the SDE along with locality aware hashing techniques can reduce the computational load when, for instance, operations requiring pairwise comparisons among streams are engaged in each workflow.
- Federated Scalability: in settings composed of several geographically dispersed computing clusters or clouds, utilizing synopses in each of these and communicating synopses instead of full local streams reduces the communication cost of global (over the union of streams arriving at the various clusters) operator evaluation.

Second, the SDE and the library of data summarization techniques it includes can serve as tools provided to the INFORE Optimizer Component so that the latter can perform synopsis-based optimization on INFORE workflows, i.e., if the application has declared that it can tolerate approximate results to a submitted workflow, the Optimizer can replace exact operators engaged in the workflow, with equivalent, approximate ones so as to speed up the processing under certain accuracy constraints.

Figure 25 below illustrates the internal architecture of the SDE Component. The details of the architecture and the SDE library are included in deliverable D6.1 submitted on M12 of the project as well. We will here elaborate on the SDE API, used by other INFORE Components as well as upstream (providing input) or downstream (receiving input) operators of a workflow engaging data synopses. The proof-of-concept implementation of the SDE Component is developed in Flink and Kafka, while the SDE library is implemented in Java. The SDE is to be provided as a continuously running service to different, currently executed/submitted workflows. Therefore, it can simultaneously maintain multiple synopses used in a variety of workflows.

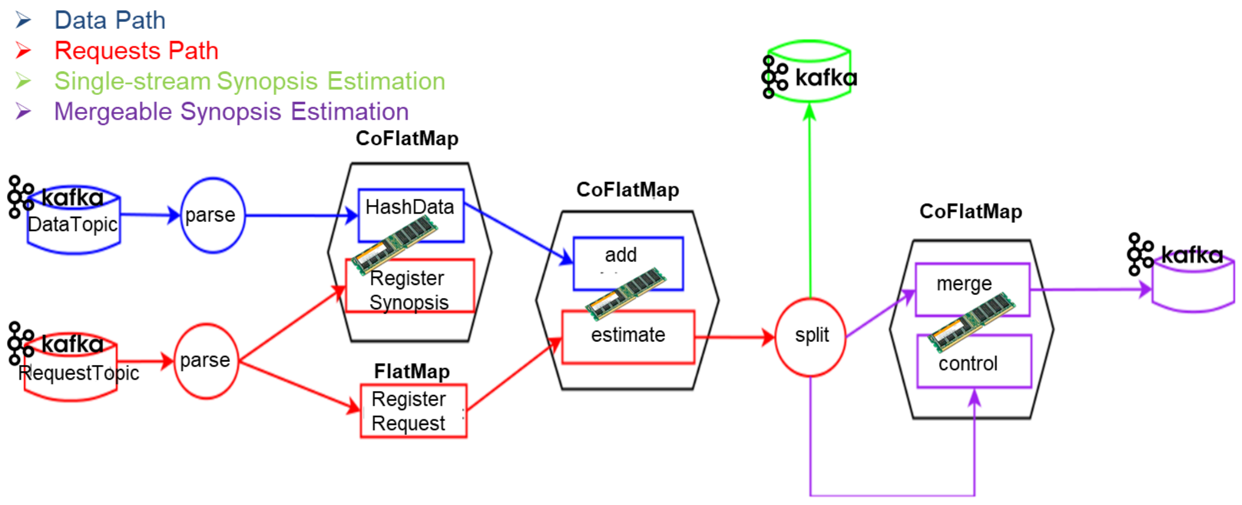


Figure 25: Illustration of the internal architecture of the SDE Component.

As shown in Figure 25, the whole inbound and outbound communication with the SDE is achieved via Kafka. All data tuples updating one or more currently maintained synopses arrive at a single DataTopic in Kafka. Moreover, all requests towards the SDE, arrive to it in a single RequestTopic in Kafka. In both cases, data tuples arriving at each topic are parsed internally by the SDE to extract information about the synopses they involve. The producers of the DataTopic and the RequestTopic are upstream operators of currently running workflows, while the consumers are the respective parsers. Finally, the results of queried synopses are streamed through one or more OutputTopics in Kafka. The producers of the output topic(s) are internal Flink operators delivering approximate query answers at the final stage of the SDE processing pipeline, while the consumers are downstream operators of running workflows.

The SDE API which is essentially implemented by the parsers consuming respective Kafka messages in Figure 25 provides the following facilities:

- Build/Stop Synopsis Request: a synopsis can be created or ceased on-the-fly, as the SDE is up and running. In that, the execution of other running workflows that utilize synopsis operators is not hindered. A synopsis may be (a) a single-stream synopsis, i.e., a synopsis (e.g. sample) maintained on the trades of a single stock, or (b) a data source synopsis, i.e., a synopsis maintained on all trades irrespectively of the stock. Moreover, the API allows submitting a single request for maintaining a synopsis of the same kind, for each out of multiple streams coming from a certain source.
- Load Synopsis Request: the SDE library incorporates a number of synopsis operators, commonly used in practical scenarios. The Load Synopsis facility supports pluggability of the code of additional synopses at runtime, their dynamic loading and maintenance at runtime.
- Ad-hoc Query Request: the SDE accepts one-shot, ad-hoc queries on a certain synopsis and provides respective estimations (approximate answers) to downstream operators or application interfaces, based on its current status.

- Continuous Queries: continuous queries can be defined together with the request for building a synopsis and they provide an estimation of the approximated quantities, such as counts, frequency moments or correlations, when the synopsis is updated due to reception of a new tuple.
- SDE Status Report: the API allows querying the SDE about its status, returning information about the currently maintained synopses and their parameters. The purpose of this facility is two-fold. First, it is useful during the definition of new workflows, since it allows the application to discover whether it can utilize already maintained data synopsis. Second, such information is useful to the Optimizer Component which, given a workflow and an accuracy budget attempts to speed up the processing and harness memory utilization by replacing exact operators (e.g. for cardinality estimation) with equivalent, approximate ones. Here we note that this facility does not involve following some of the processing paths shown in Figure 25, but instead augmenting the QueryableState part of Flink's DataStream API, so that it provides synopsis-specific information at runtime.

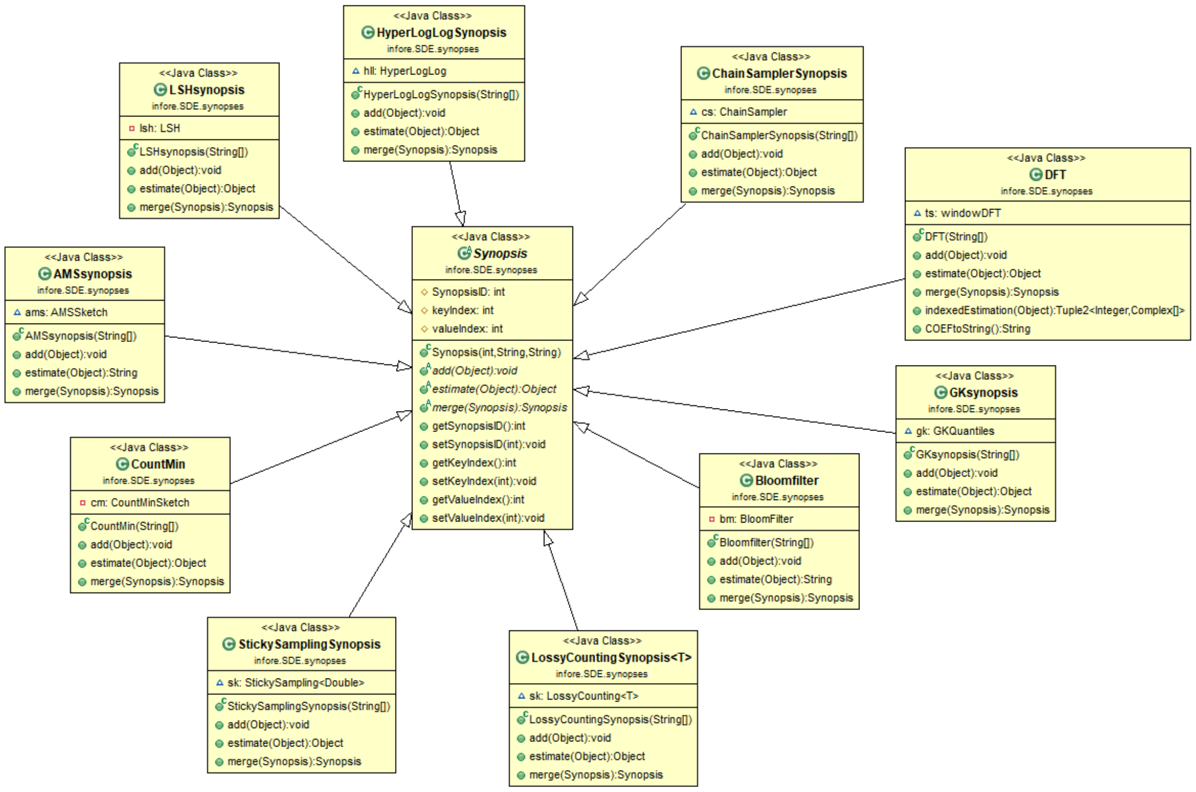


Figure 26: Overview of the SDE library.

As shown in Figure 26, the SDE library leverages subtype polymorphism to ensure the facile pluggability of new synopses definitions. In a nutshell (please see deliverable D6.1 for further details), a generic Synopsis class is extended by classes implementing algorithms of specific data summarization techniques, and their processing methods override those of the parent class.

8.1.3.6 Complex Event Forecasting Component

The Complex Event Forecasting Component provides algorithms for complex event processing and complex event forecasting. Like the Synopsis Data Engine Component, the Complex Event Forecasting Component consist of a library providing the algorithms and methods for complex event processing and forecasting and operators containing the configurations and information necessary to execute these algorithms and methods.

The physical implementation of operators enables the users of the INFORE Architecture to perform complex event processing and forecasting on their streaming data. These physical implementations materialize the respective Logical Operators the users include in a designed workflow by simply drag and dropping them using the Graphical Editor Component.

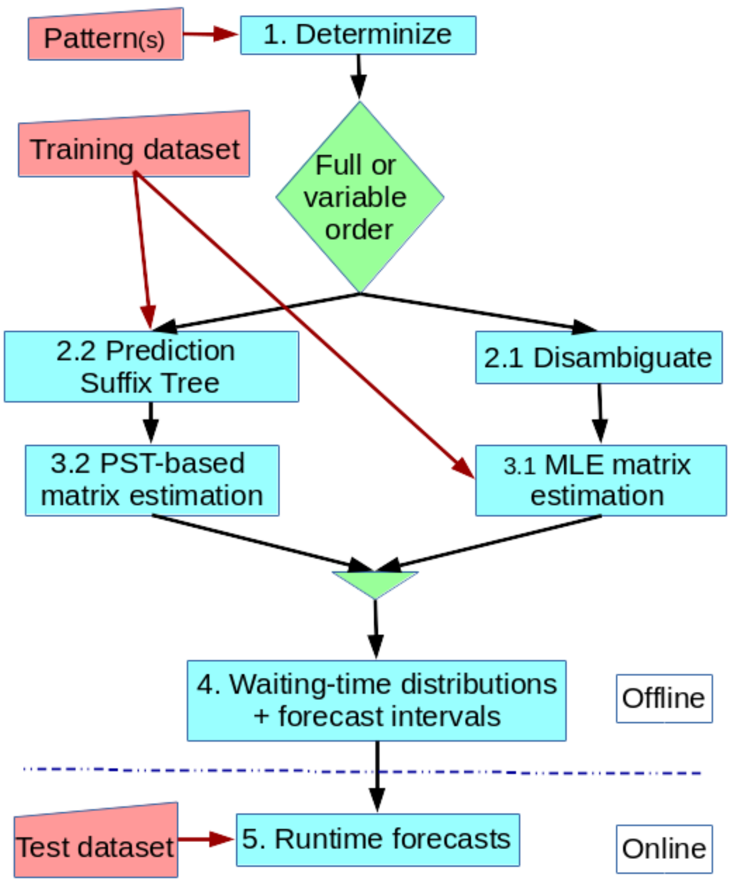

Pattern(s)
1. Determinize
Training dataset
Full or variable order
2.2 Prediction Suffix Tree
2.1 Disambiguate
3.2 PST-based matrix estimation
3.1 MLE matrix estimation
4. Waiting-time distributions + forecast intervals
Offline
Test dataset
5. Runtime forecasts
Online

Figure 27: Internal architecture of the Complex Event Processing and Forecasting (CEP/F) Component.

The internal architecture of the Complex Event Processing and Forecasting (CEP/F) Component is shown in Figure 27. The various modules of the CEP/F Component will be described in more detail in deliverable D6.3. Here we provide an overview at a higher level of abstraction.

Before the CEP/F Component can start consuming streaming data (a “test dataset”) and produce forecasts in an online manner, it must first go through several offline steps. These offline steps require two input types: a) First, a set of patterns/queries with which a user needs to monitor streams of input events. These patterns are expressed in the form of symbolic regular expressions, i.e., regular expressions whose terminal symbols are not simple characters but Boolean expressions. For simplicity, we assume that a single pattern is provided. If multiple patterns are provided, the same steps must be repeated for each pattern. b) Second, a training dataset, representative of the streaming dataset to be encountered in runtime. This training dataset is used in order to build a probabilistic model for the pattern. Based on this model, the actual forecasts will be built.

The first step (1. Determinize) is to use the symbolic regular expression of the pattern in order to construct an automaton that will act as a computational model for it. If we are interested only in CEP (and not forecasting), then the rest of the offline steps may be skipped. The automaton can be used to detect complex events by directly consuming a stream of input events. From an architectural point of view, it is important to note that the automaton itself acts as a prototype/template for continuously spawning multiple automaton runs that actually consume the input events. For example, in the maritime use case, a pattern may need to be applied on a per vessel basis. In this case, for each new vessel, a new automaton run is created that will be responsible for this vessel.

If the goal is to perform forecasting, then there are two paths that can be followed in order to build the required probabilistic model (path 2.1-3.1 and path 2.2-3.2). In both cases, the goal is to derive a description of a pattern's automaton in the form of a k-order Markov chain. The right-hand path in Figure 27 (2.1-3.1) corresponds to a straightforward method for deriving such a Markov chain, whereas the left-hand path (2.2-3.2) to an optimized method that allows k to reach higher values. This Markov chain essentially describes how the automaton can move among its various states. For each such state, we can use the Markov chain to predict how the automaton will behave (which states it will visit) and when it is expected to reach a final state and thus detect a complex event (step 4). We currently assume stationarity. As a result, for the automaton to function as a forecaster in an online fashion, all we need to do is to enrich each of its states with a forecast (the Markov chain itself may be dropped after the forecasts have been estimated). The engine then works in a manner like that for CEP, with the difference that each automaton run, besides a reference to its prototype, also has a reference to a lookup table of forecasts.

The CEF module will communicate with the rest of the architecture via Kafka. For the CEF to function properly, two upstream and one downstream Kafka topics are required. As described above, the CEF will function in a two-phase manner, according to requests coming from the upstream topics: a) an "offline", training phase that will construct the necessary probabilistic models for a set of patterns, and b) an online testing phase, where the previously constructed models are used to actually generate forecasts. The three Kafka topics are the following:

- A ConfigTopic. Through this topic, the CEF module receives requests for training and testing. A training request must be accompanied by a set of patterns (along with their orders and partition attributes) and a set of declarations. A testing request must be accompanied by values for the parameters of confidence threshold, maximum spread and horizon. For an explanation of these parameters and concepts, please consult deliverable D6.3. This topic is also used for stopping a running CEF process, as well as for querying the CEF module for its status and its various parameters.
- A DataTopic. Through this topic the CEF module receives both the training and testing datasets, in the form of streams of events (tuples). As soon as a training/testing request arrives at the ConfigTopic, the CEF module starts consuming data from the DataTopic to either train its models or produce forecasts.
- A ForecastTopic. This topic is essentially used only in the testing phase. This is where all Complex Events and Forecasts detected and generated by the CEF module are written.

The Complex Event Forecasting methods are described in more detail in the deliverables D6.2, D6.4 and D6.5.

8.1.3.7 Interactive Online Machine Learning Component

The Interactive Online Machine Learning Component provides tools for interactive online machine learning. Like the Synopsis Data Engine Component and the Complex Event Forecasting Component, the Interactive Online Machine Learning Component consists of a library providing the algorithms and methods for its purpose and operators containing the configurations and information necessary to execute these algorithms and methods.

The operators enable the users of the INFORE Architecture to perform interactive online machine learning on their streaming data, by simply drag and dropping them in the streaming analysis process, which is designed by using the Graphical Editor Component.

The Online Machine Learning Component learns expressive and interpretable complex event patterns from streaming input of time-stamped information. It functions in an online fashion, i.e., it continuously makes predictions (detects complex events of interest) on incoming data, using the labelled fragments of the data as feedback, from which it updates its current model (event pattern set), thus improving both its predictive performance and the quality of the learnt event patterns over time.

The learnt patterns have the form of weighted logical rules, and the learning algorithm is capable of both inducing their structure (the actual rules) and optimizing their weights. Together, the rules and their weights define a probabilistic predictive model that is resilient to noise and uncertainty. The learnt patterns may subsequently be used for complex event recognition & forecasting, while, thanks to their interpretability, they may also be used by human experts for acquiring novel insights about the application domain via simple inspection. In addition to learning event pattern sets from scratch, the Online Machine Learning Component is also capable of revising existing pattern sets (e.g. an initial, potentially crude set of patterns provided by domain experts) from new data that stream-in.

The Online Machine Learning Component will communicate with the rest of the architecture components via Kafka. For this, two upstream and one downstream Kafka topics are required. These Kafka topics are as follows:

- A ConfigTopic (upstream): Through this topic, the Online Machine Learning Component receives a configuration of its learning algorithm's hyperparameters: a learning rate (Double), a regularization rate (Double), a Hoeffding test statistical confidence threshold (Double), a pruning threshold (Double) and a loss function name (String). A detailed description of these hyperparameters will be included in deliverable D6.2. Also, via the ConfigTopic, the learning algorithm receives a set of declarative syntactic specifications for synthesizing rules from the encountered data, an application-specific form of domain knowledge, which will also be detailed in deliverable D6.2. An initial event pattern set, potentially provided by domain experts, and constantly under revision from that point on, may also be provided through the ConfigTopic.
- A DataTopic (upstream): Through this topic the Online Machine Learning Component receives its input data in the form of a stream of event tuples. The Learner continuously listens to this topic and consumes data that arrive there, to first make predictions with its current event pattern and then use any potential labels in the data to update the event pattern set.
- A LearningResultsTopic (downstream): The Online Machine Learning Component outputs its results to this topic. There are three types of information that are output here: (i) the actual predictions of the learner, i.e., the complex events it detects from its input stream; (ii) online learning statistics, such as the learner's evolving online error rate and prequential F1-score on the input data and mean CPU processing time per input data point over time; (iii) the learner's evolving model (event pattern set) over time.

The Interactive Online Machine Learning methods are described in more detail in the deliverables D6.2, D6.4 and D6.5.

8.2 Section 5 of Deliverable D4.1: Technical Constituents of Streaming Analysis Workflows

This section is from the deliverable D4.1 and describes the technical constituents of streaming analysis workflows, as defined in the D4.1 deliverable. As the initial prototype, described in this deliverable D4.2, is the implementation of the Architecture, this section is added for convenience.
The next sections cover the actual implementation details of the architecture components and the definition of the interfaces between them.

8.2.1 Realization of the INFORE Architecture

The conceptual design of the INFORE Architecture has been described in Section 4 (of Deliverable D4.1). In this section, we describe the concrete realization of its components.

The project partner RapidMiner provides RapidMiner Studio, an open-source software solution for the Connection Component and the Graphical Editor Component. The existing user interface is enhanced to implement the connections to the other components of the architecture. RapidMiner Studio offers already a graphical representation of an analysis process, by providing operators which can be placed and connected with Drag & Drop into a process design GUI. This allows us to hide the complexity interacting with different technology stacks from the user, by providing a common usage concept. At its current state, the Studio does not provide support for the functionality envisioned by INFORE and, thus, will be significantly extended in the scope of the project.

In particular, INFORE adds a so-called **Streaming Nest** operator to RapidMiner Studio. The Streaming Nest operator is a subprocess operator, which means that a family of operators can be placed inside of it. Streaming operators are placed inside the Streaming Nest operator and are connected during workflow design time to define the data flow of the streaming analysis process. Hence the subprocess of the Streaming Nest operator is the implementation of the Graphical Editor Component. Synopses Data Engine, Complex Event Forecasting and Interactive Online Machine Learning Logical Operators are added by INFORE, which are implementing the operator part of the corresponding components (see Sections 8.1.3.2, 8.1.3.5, 8.1.3.6 and 8.1.3.7). These "INFORE Component" operators contain all information to execute the specific algorithm and methods, which are provided by corresponding libraries. Figure 23 shows RapidMiner Studio, illustrating the drag and drop approach of the process design. The Streaming Nest operator and its subprocess are also demonstrated, as well as some example streaming and "INFORE Component" operator. Again, the front end allows us to present an interactive workflow design to the users, while the underlying complexity is hidden from them. In the case of multiple available platforms, either the user can simply manually choose which platform to use, or call the Optimizer Component for an optimized workplan (see Section 8.1.3.4).

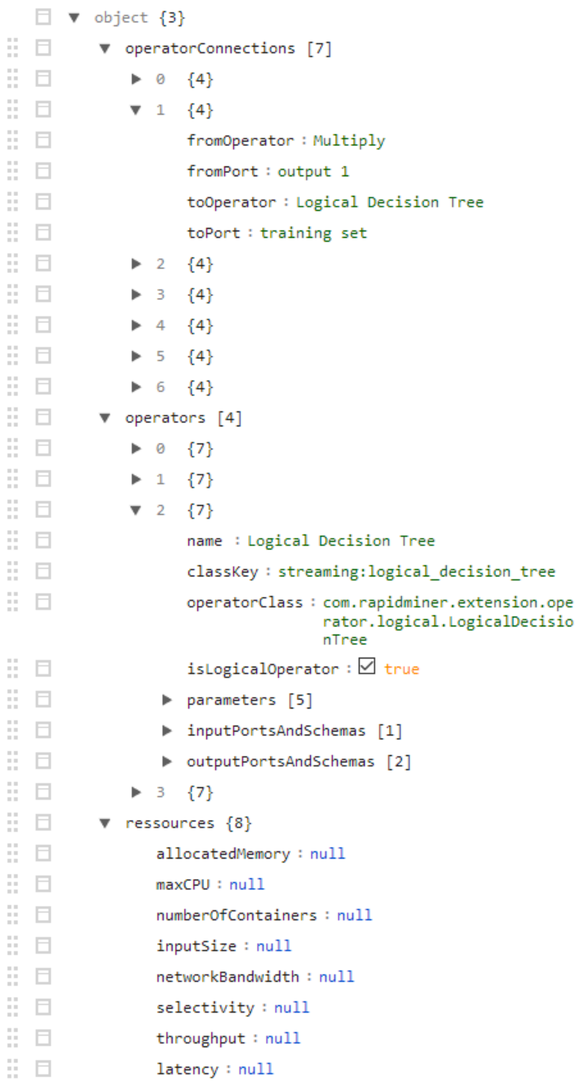
RapidMiner Studio also provides the concept of Connection objects. Thereby all information to connect to a specific system (for example a database) are packaged in an object inside RapidMiner Studio. This object can be stored in the RapidMiner Repository (the data and process storage system of RapidMiner Studio) and can be utilized in an analysis process by dragging and dropping it in the process design GUI. RapidMiner Studio also offers the possibility to easily create and configure these Connection objects. User management handling and the secure injection of critical information (e.g. passwords) are also provided by RapidMiner Studio. More information about the Connection Management concept[3] of RapidMiner Studio is available. INFORE adds Connection object classes for all supported streaming backends (see Section 8.2.2), thereby implementing the described Connection Component (see Section 8.1.3.1). This is essential to support the cross-platform optimization of data stream analysis in INFORE.

Moreover, NFORE adds a Java library, implementing the functionality needed for the Manager Component (see Section 8.1.3.3). This Manager Component library provides capabilities to connect and receive resource information from a streaming computing cluster, to connect and consume data streams, to execute and deploy streaming analysis process and to produce data streams as an output. These capabilities are explained in more details in the following sections.

As an interface to the other INFORE Components, especially for the interface between Manager Component and Optimizer Component, a JSON representation of the streaming analysis process is added by the INFORE project. A ProcessToJSON converter class is implemented in the Streaming Nest operator. The inputs of this ProcessToJSON converter are the designed streaming analysis workflow and the resource information of the streaming processing backend. The first is provided by the Streaming Nest operator, the second is retrieved by utilizing the Connection object to connect to the streaming processing backend(s) and retrieving the corresponding information. The ProcessToJSON converter combines these two information sets and converts them to a JSON representation, which is provided to the Optimizer Component as an engine-agnostic workflow.

Engine-agnostic workflows are encoded into an AgnosticWorkflow class that consists of three core variables: Operators, OperatorConnections, and Resources. These represent the operators of the workflow, the connections among them, and the resources allocated to the workflow, respectively. An example JSON representation of a simplistic workflow is depicted in Figure 28. This example shows an operator named "Logical Decision Tree" that gets its input from port "output 1" and propagates its result to the port "training set". In this JSON description, the operator is a logical operator (isLogicalOperator=true). The operator description also specifies implementation details such as the class that implements this operator.

[3] https://docs.rapidminer.com/latest/studio/connect/#connection-objects/



```
▼ object {3}
   ▼ operatorConnections [7]
      ▶ 0  {4}
      ▼ 1  {4}
           fromOperator : Multiply
           fromPort : output 1
           toOperator : Logical Decision Tree
           toPort : training set
      ▶ 2  {4}
      ▶ 3  {4}
      ▶ 4  {4}
      ▶ 5  {4}
      ▶ 6  {4}
   ▼ operators [4]
      ▶ 0  {7}
      ▶ 1  {7}
      ▼ 2  {7}
           name  : Logical Decision Tree
           classKey : streaming:logical_decision_tree
           operatorClass : com.rapidminer.extension.ope
                           rator.logical.LogicalDecisio
                           nTree
           isLogicalOperator : ☑ true
         ▶ parameters [5]
         ▶ inputPortsAndSchemas [1]
         ▶ outputPortsAndSchemas [2]
      ▶ 3  {7}
   ▼ ressources {8}
        allocatedMemory : null
        maxCPU : null
        numberOfContainers : null
        inputSize : null
        networkBandwidth : null
        selectivity : null
        throughput : null
        latency : null
```

Figure 28: Example JSON representation of a streaming analysis workflow.

Table 2 gives an overview of the different fields and their meaning in the engine-agnostic workflow representation.

Key	Description	Key	Description
Operator Connections		isInputPort	Indicator if this is an input port
fromOperator	Name of the source operator	isConnected	Indicator if port is connected
fromPort	Name of the source port	schema	If object is a data set, schema of this data set
toOperator	Name of the sink operator	**Schema**	
toPort	Name of the sink port	fromMetaData	Indicator if schema is retrieved from meta data
Operator		size	Size of the data set at the port
name	(unique) name of the operator	attributes	List of Attributes
classKey	RM specific key for the operator class	**Attribute**	
operatorClass	Name of the java operator class	name	Name of Attribute
isLogicalOperator	Indicator if the operator is a logical one	type	Value type
parameters	List of Parameters (see below)	specialRole	Special role
inputPortsAndSchemas	List of input ports and their schema (see below)	**Resources**	
outputPortsAndSchemas	List of output ports and their schema (see below)	allocatedMemory	Allocated memory for the process
Parameter		maxCPU	Maximum number of CPU for the process
key	Key of the parameter	numberOfContainers	Number of containers used
value	Current parameter value (as String)	inputSize	Size of the input data
defaultValue	Default value (as String)	networkBandwidth	Available network bandwidth
range	String representation of the range	selectivity	Indicator of the selectivity
typeClass	Name of the Java parameter class	throughput	Current throughput
PortAndSchema		latency	Current latency of the network
name	Name of the in- or outputport		
objectClass	Name of java class of the object delivered or received at this port		

Table 2: Overview of the different fields and their meaning in the engine-agnostic workflow representation.

One of the facilities of the Manager Component is to convert or package one or more sub-sets of operations of the streaming analysis workflow (or the whole workflow) into a deployable artefact, which can then be deployed on one or more compute backends. The implementation of this feature is provided as part of the Manager Component's deployment functions (see Section 8.1.3.3). From the graphical modelling point of view, this can be achieved in a couple of ways. In the first scheme, the Optimizer may decide (during optimization of workflow) to place a certain group of operators as single or multiple deployable artefact(s) on a certain backend, and actuate on this decision, by invoking the relevant functions of the Manager Component.

In the other scheme, the human designer who creates the graphical workflow may indicate a set of operations as a single deployable artefact by grouping them. For example, this may be the case when one compute backend (say a Flink cluster) is available (or preferred). In this case, the logical operators can be configured to use Flink-based concrete implementations for data preparation, transformation, modelling and other operations. With this knowledge, the human designer may configure the NEST operator to: i) not rely on Optimizer for placement decisions or ii) indicate to Optimizer to respect the indicated grouping, or even iii) let the Optimizer overrule these groupings (placement indicators) if it deems fit. This concept will be further refined as the prototyping work progresses, but the idea is to aim at maximum flexibility for the end users.

In the following section, we briefly describe the technical constituents of an INFORE streaming analysis workflow, and how INFORE processes abstract over the various streaming Big Data platforms.

8.2.2 Producers, Consumers and Compute Clusters

As explained in Section 8.1.3, the INFORE streaming analysis workflow is composed of input stream(s), the streaming analysis workflow and potentially output stream(s). The input operators of the analysis workflow are sometimes referred to as downstream or input receiving operators. The output operators are sometimes referred to as upstream or input providing operators. Entities which generate the stream and consume it are referred to as Producers and Consumers, respectively. Computer clusters refer to the stream processing backend, which are often used to deploy the stream processing workflow so complex computation is performed in a scalable manner on dedicated resources.

8.2.2.1 Establishing a reliable, robust, generic and flexible interface between disparate Components

The added value of INFORE is that its streaming analysis workflows follow an abstraction approach that allows them to span multiple streaming Big Data platforms for producers, consumers and compute clusters for execution. INFORE delivers this flexibility by performing the heavy lifting needed to integrate with different backends as part of its components may run on different Big Data platforms. Some of the biggest technical challenges faced here are listed below in terms of feature requirements:

- **Interfacing heterogenous or disparate components**: Providing a high level of abstraction is a huge challenge because a mechanism needs to be devised that would allow INFORE Components to communicate in a *reliable* and *robust* manner with possibilities to incorporate future components, which are themselves expected to be disparate and available on heterogenous backends. An acceptable solution should allow to provide a *generic* interface, which is easy to adopt across the different tiers of the platform.
- **Interfacing stream sources and sinks**: Another hard requirement is to let a variety of producer and consumer streams to be integrated. The schema of the data tuples is not always pre-defined or known but can be analysed at design time. Hence, it is highly desirable to use a *generic communication middleware* with support for simple and complex data structures. A feasible solution would not just make the sources and sinks (implemented in any language) available over a standard interface, but also the flexibility to deal with different data (tuple) types, a criteria to slice and dice different windowing of streams and ideally also support some degree of persistence and querying.
- **Dispatching and deploying streaming analysis workflows:** Dispatching (deploying) streaming analysis workflows to available compute cluster or a compute cluster of choice is a task which often requires to communicate with a specific Big Data platform using its custom APIs or client utilities, but here again INFORE provides a generic layer within the Manager Component (in principle, similar to Apache jClouds[4] library that allows to create applications that are cross-cloud portable).

Keeping these requirements in sight, it becomes vital to adopt a state-of-the-art communication middleware for INFORE. This middleware is expected to deliver above requirements and would serve at different tiers of the INFORE architecture to communicate with internal components (Synopses Data Engine, Complex Event Forecasting), various sources and sinks as well as assist in fetching results from ongoing experiments on the HPC (e.g. in the Life Sciences use case). Overall, this messaging and communication middleware must allow INFORE system to function seamlessly and detach the sender and receiver and receiver sides through a messaging queue, which serves to pass data as topics or digests. Thus, the choice of this middleware required a comparative analysis of the leading related technologies including RabbitMQ, ActiveMQ, Flume and Kafka to name the top candidates. A short overview of this comparative analysis is presented next.

8.2.2.1.1 Comparison of leading communication middleware technologies

[4] https://jclouds.apache.org/

ActiveMQ
Apache ActiveMQ[5] is a Java-based multi-protocol messaging server, which allows to integrate multi-platform applications using the Advanced Message Queue Protocol (AMQP). In addition, it also supports other open wire protocols like OpenWire, Stomp and MQTT. It is more suited for transmitting data in binary rather than text-based format. Its server side consists of Master broker nodes and a set of slave nodes, which can be paired together so that data can be provided efficiently to consumer processes, address fault-tolerance and move data between nodes. ActiveMQ can be setup as an embedded application with a small footprint, which then serves as a messaging endpoint for inter-application communication.

Although ActiveMQ allows for applications written in different languages to pass messages asynchronously, in its current implementation, its feasibility in high-throughput and transmission of large data streams is not the best. There is no easy way to batch messages together and it is assumed that it uses a batch size of 1. Experiments conducted on streams of logging data **[2]** revealed that the persistence mechanism in ActiveMQ consumed 70% more disk space than Kafka to store the same set of 10 million messages. The lag in performance at the persistence tier is attributed to the usage of JMS (Java Messaging System) specification which requires a thorough message header. The server processes seem to get hogged into maintaining the persistence indexes, that require B-Tree instance(s) to maintain metadata and state of each message. On the functional side, ActiveMQ is not intended for performing stream processing operations like windowing, aggregations or groupings on messages. Hence, its adoption would not satisfy most requirements of INFORE architecture and use cases.

RabbitMQ
RabbitMQ[6] is a lightweight and widely deployed message brokering system implemented in Erlang. Like ActiveMQ, it also supports several messaging protocols including AMQP, Stomp and MQTT. The server side of RabbitMQ supports consists of a cluster of nodes, which provide high availability and data replication at the persistence tier, for messages exchanged between producers and consumers. Connections from clients, the channels and queues used to pass data, are distributed across the nodes, to offer an efficient and scalable handling of data. In this aspect, RabbitMQ make up a formidable candidate technology for INFORE’s communication middleware.

However, despite some interesting features, RabbitMQ is not a technology that is targeted for high throughput data stream processing as required by various INFORE use cases, intra-component interaction and interfacing between operators of INFORE’s streaming analysis workflows and INFORE components. A performance evaluation of RabbitMQ regarding the production and consumption of high-throughput messages yielded results similar to ActiveMQ and inferior to Kafka. In **[2]** , Kafka is also compared with RabbitMQ. Kafka could produce up to 50,000 messages per second for a batch size of 1, which was two times higher than RabbitMQ. Configuring larger batch sizes in RabbitMQ is also not obvious. When message consumption is evaluated, Kafka performed four times faster than RabbitMQ (and ActiveMQ) by consuming up to 22,000 messages per second. This overhead is associated with the fact that RabbitMQ maintains the delivery state of each message, while Kafka does not. Finally, RabbitMQ requires the setup of Erlang execution environment on the machines. As Erlang is not so widely available or platform independent (unlike Java), the provisioning of Erlang runtime poses an additional requirement. Overall, RabbitMQ is not the best fit to be adopted as INFORE’s messaging and communication middleware.

Flume
Apache Flume[7] is a project that provides a simple and flexible architecture based on streaming data flows. A Flume system is made up of Flume agents, which are Java processes. A Flume agent acts as a middle-man between the producer and consumer of data. The agent internally has a source component, a sink component and a channel in which data received by the agent is placed via the source component. The source component receives data from an external producer. The sink component reads the message from the channel and passes it out to an external consumer. The internal channel is backed up by a data store. A Flume agent can be configured in a multiplexing manner, i.e. it can maintain multiple channels which pass on the data to one or more external consumers (such as an

[5] https://activemq.apache.org/
[6] https://www.rabbitmq.com/
[7] https://flume.apache.org/index.html

HDFS cluster or a Kafka topic). Flume can be used for interaction between different applications or components that need to exchange or collect data. Flume agents can be horizontally scaled. Flume guarantees reliable delivery of a message, which is called as an event in Flume. Although Flume provides an interesting data flow management middleware, it is rather intended for scenarios, which need to push high throughput data from many sources into a collector sink, via a data-lake system that may perform transformations on passed data.

From the INFORE perspective, Flume has some major shortcomings. Flume does not offer a persistence tier as part of its framework. This implies that data in channels is not replicated across the Flume agents. Flume only provides weak ordering on transmitted messages, and duplicate messages are often sent due to its emphasis on reliable durable delivery, but which need to be cleaned at the consumer side, especially given their ability to induce noise. Flume's capabilities are further undermined due to its more complex management overhead, which can affect its throughput handling, scalability and reliability aspects if the data stores backing the channel(s) are not appropriately configured for all agents [1] . Due to its rather limited and very data-lake specific features, Flume is not a candidate for adoption in INFORE.

We conclude our comparative analysis with Kafka, which we present in the next dedicated section, to highlight how it is a clear winner to be adopted as our *communication middleware of choice* for INFORE.

8.2.2.1.2 Kafka - Communication middleware of choice

Kafka is a highly scalable and fault tolerant distributed data stream platform. It provides persistent storage on the server side, which is based on a cluster of so-called broker instances. The brokers provide data storage and replication by means of partitions in the broker. This leads to high availability of data and acts as a scalable tier that can deal with large number of producers and consumers - without causing contention at the network or storage level access. Due to these properties, Kafka plays a central role in the INFORE Architecture as its communication middleware of choice. It is employed due to its flexible API- based features, a highly performant backend that can deliver high throughput, large-scale data streams with reliability, strong ordering guarantees, replication-capable and scalable server-side. These features are able to meet data exchange and interfacing requirements of INFORE use cases and components. Hence, in the following, we describe how Kafka usage in INFORE Architecture spans across the board - from integrating different components and interfacing INFORE's streaming analysis workflow (operators) with different producers and consumers (sources and sinks).

As noted, initial prototyping efforts could leverage Kafka for bridging disparate components, which are distributed over different locations. In particular, in Sections 8.1.3.5, 8.1.3.6 and 8.1.3.7 we showcased that the Synopses Data Engine, the Complex Event Processing and Forecasting, as well as the Interactive Online Machine Learning Components of INFORE are interfaced with upstream and downstream operators via Kafka topics. Thus, treating a "topic" as a medium for asynchronous communication, applications written in any language and having complex dependencies can be loosely coupled using a clear interface and minimal effort. Kafka topics are at the heart of its publish/subscribe mechanism, that can send and receive continuous streams of data. Kafka producers write data or data streams to topic(s), which may be persistently stored on the brokers, and the consumers retrieve them via topic-based access.

Consumer access is also highly scalable and can be parallelized leveraging the same principle of clustered brokers and replicated partitions at the server side. A typical Kafka program (represented as a processor topology – see description of Stream API below) can use multi-threading to achieve concurrent execution, which improves performance. In this way, Kafka provides a technology that can pass data streams efficiently from producers to consumers and allows to write stream processing workflows that can be tuned for high performance and scalability.

Kafka provides a set of APIs, which are worth mentioning briefly:

- Connect API: This API allows to write connectors for bringing data from a source into the Kafka system and to bring data out from Kafka into a sink system.
- Producer API: This API can be used by Kafka applications to create topics and send data streams to topics.
- Consumer API: This API allows to receive data streams from Kafka topics into a Kafka application.

- Stream API: This API allows to process data streams from input topic(s), which transforms them to output stream(s), which can be written to other topic(s). A computation step (transformation) is referred as a processor node. The configuration of these nodes is called a processor topology.
- Admin Client API: This API allows to manage topics, brokers, access control lists and other objects on the Kafka server side.

To integrate Kafka in the INFORE streaming analysis workflows, the Manager Component functions and the Graphical Editor Component (operators) would be implemented to make use of Kafka APIs. Together with the Connection Component object, the operators for performing transformations enable the following capabilities:

- Connecting with the Kafka backend, creating topics, sending and receiving data streams to and from topics.
- Performing typical transformations on incoming data streams. This includes windowing, joining, aggregation and grouping, filtering, computing some statistics, mapping the tuple(s), training or updating models, applying models, etc.

These examples illustrate how Kafka plays an important technical role at different levels of the INFORE architecture by simplifying the complexity of integrating legacy, heterogenous or niche systems, helps standardize several interfaces in INFORE and provides a central communication technology for connecting the different components in the INFORE Architecture.

In the following section, we provide a short description of how INFORE deals with the requirement of dispatching (deploying or placing) streaming analysis workflows on compute clusters or Big Data processing platforms. The list of supported platforms is not fixed. The idea is to extend the support to more platforms in a gradual fashion.

8.2.2.2 Incorporating Big Data Platforms for Dispatching Streaming Analysis Workflows

As a reference on how Big Data platforms are incorporated in INFORE after the Optimizer Component selects a specific platform for dispatching or deploying a part of a streaming workflow, we take Flink for a more elaborate discussion. Besides streaming Big Data platforms, INFORE would also incorporate support for HPC infrastructures as the development progresses.

Flink provides a state-of-the-art implementation of streaming analysis functionality. It is also used as the implementing technology for components of the INFORE Architecture such as the Synopses Data Engine. Hence it is a very good demonstration of integrating a streaming analysis technology into the INFORE Architecture. In the following we describe how the Flink is integrated into the INFORE Architecture.

Flink consists of several distinct components that interact with each other to provide a functional and scalable streaming environment. The server side of Flink comprises mainly of the Job Manager, Resource Manager and Task Manager components.

The *Job Manager* is responsible for the execution of a single application. The manager transforms the application data and workflow description into an executable unit and requests the required resources. The *Task Managers* provide the slots for the actual execution. The slots inside each task manager run as threads in the same JVM instance, while several task managers can work on the same application, thus sharing data between separated JVMs.

In addition, the *Resource Manager* distributes requests of the *Job Managers* to the *Task Managers*. It can work with external resource managers such as YARN, or as a standalone deployment. Depending on the amount of parallelism and resources available, a single job can be deployed on multiple *Task Managers* and Flink can rescale running jobs.

Flink offers two different styles for deployment: framework and library. The framework deployment bundles the application into a JAR file and hands it over to a running service, like a resource manager or directly to a Flink Job Manager. This way an application can be directly planned for execution (in case of the submission to a Job Manager) or scheduled to hand over to a Job Manager (e.g., via YARN or a Flink dispatcher).

The library style uses two independent Docker images: one for bundling the application (including Job- and Resource Manager) and one for running the Task Managers. The framework deployment is a way of submitting an application via a client to an already running service. For large scale infrastructure deployments, this approach is probably easier to maintain and set-up. While the library style is a bit more flexible and more suited for a microservice oriented approach.

In regard of the INFORE architecture both styles can be easily integrated and do not differ too much. The framework style is the one that can, in combination with an existing resource manager, be deployed in existing cluster settings where additional task managers can spin up on demand and the processes can easily be scaled out. The deployment as a library is useful for rapid prototyping and testing out applications in a local setting.

Flink provides two core APIs, namely the DataStream API and the DataSet API. The DataStream API allows us to manage bounded or unbounded streams of data and the DataSet API allows to manage bounded data sets. Flink also offers a Table API, which is a SQL-like expression language for relational stream and batch processing that can be easily embedded in Flink's DataStream and DataSet APIs.

For integrating Flink capabilities in INFORE, the functions of the Manager Component and the operators of the Graphical Editor Component will be implemented to wrap various features of the Flink APIs. Under the hood, this would allow to create a Flink program that can be deployed on the Flink cluster. A Flink program can perform various transformations on the data stream. These transformation operators are combined into a sophisticated topology called a streaming dataflow.

The execution of this dataflow is inherently parallel and supports distribution. Flink performs various execution-time optimizations on the streaming dataflows. During execution, a stream is divided into one or more stream partitions, and each operator is logically divided into one or more subtasks. The operator subtasks are independent of one another and can be executed in different threads. For distributed execution, Flink chains operator subtasks together into tasks and each task is executed by one thread. Chaining operators together into tasks is a useful optimization as it reduces the overhead of thread-to-thread handover, buffering and increases overall throughput while decreasing latency.

Due to these capabilities, Flink plays an important role in the INFORE architecture. It is currently used for implementing components or incorporating them as a streaming application. These components include the Synopsis Data Engine (SDE), a version of the parameter server for machine learning operators (another one is being implemented in Akka) and a proof-of-concept for the Complex Event Processing/Forecasting Component. However, we emphasize that INFORE Architecture is not limited to Flink as a streaming Big Data platform, in contrast, INFORE's approach is broad-scoped and purpose-built for abstraction i.e., it specifically aims at incorporating multiple platforms.

8.2.3 Creation and Deployment of a streaming analysis workflow using the INFORE Architecture

In the previous chapters, we described the component-based structure of the INFORE Architecture. The provided functionality and the interaction between the components are described in detail. In addition, challenges and requirements from integrating and building a *cross-platform* framework for different streaming technologies are discussed.

We now present a step-by-step process on how a streaming analysis workflow is created, optimized and deployed using the INFORE Architecture. The different steps and interactions and how the components come into play are detailed:

1. **Design of the streaming analysis workflow:**
   The user of the INFORE Architecture designs a streaming analysis workflow. Therefore, she uses the **Graphical Editor Component** (see Section 8.1.3.2) to define the logic of the streaming analysis, without the need to take care of the technology-specific details.
   a. The user can select Logical streaming Operators, which provide an abstraction level of generic streaming analysis functionality over the streaming platforms providing this functionality. The **Optimizer Component** (see Section 8.1.3.4) later chooses the streaming backend used, optimal for the current workflow.

b. Functionality provided by the **Synopsis Data Engine Component** (see Section 8.1.3.5), **Complex Event Forecasting Component** (see Section 8.1.3.6) and the **Interactive Online Machine Learning Component** (see Section 8.1.3.7) can be leveraged through the corresponding operators in the graphical editor.
c. The user creates Connection objects for input and output streams and streaming backends (for a detailed description of the **Connection Component** and the functional concepts of input and output streams and streaming backends, see Section 8.1.3.1) through the graphical editor. The user only needs to provide essential information to connect to the streams. The Connection objects are used in the design of the streaming workflow in the same drag-and-drop manner as the streaming operators.

2. **Handover of streaming analysis workflow to Manager Component:**
When the design process of the workflow is finished (e.g., when the user presses a submit button in the extended RapidMiner Studio), the workflow is handed over to the **Manager Component** (see Section 8.1.3.3). The Manager Component converts the workflow into its JSON representation (see Section 8.2.1). It also uses the Connection objects to connect to the provided input and output streams and streaming backends and retrieves information about volume and schema of input data, available resources of the streaming engines and output streams. This information is added to the JSON representation as well. The JSON is handed over to the **Optimizer Component** (see Section 8.1.3.4).
3. **Optimization of the streaming analysis workflow by the Optimizer Component:**
The **Optimizer Component** converts the JSON representation to the tool-agnostic workflow representation. An optimization of the workflow is performed. Depending on the user-specified parameters, the optimization can include the selection of the concrete implementations of Logical Operators (providing cross-streaming-platform optimization of the workflow), the execution order and bundling of operators to specific streaming executing jobs, the insertion of synopses and function and data shipping (moving execution to the data or vice versa). More details about the optimization will be included in deliverabs D5.1, D5.2 and D5.3.
4. **Providing optimized workflow back to Manager Component:**
The Optimizer Component provides the Manager Component with the optimized workflow. This optimized workflow is visualized in the Graphical Editor Component to inform the user about the changes of the Optimizer Component.
5. **Deployment of the workflow by the Manager Component:**
The Manager Component prepares the execution of the optimized workflow by creating execution jobs defined by the bundled operators in the workflow. The streaming execution jobs are deployed on the streaming backends by the Manager Component.
6. **Monitoring and management of the deployed workflows:**
The Manager Component can be used to monitor deployed workflows. If specified by the workflow, statistics are collected and provided to the Optimizer Component, and used for displaying an overview of the running workflows to the user through the Graphical Editor Component. Running workflows can be aborted, paused, edited and resumed through the graphical user interface. If conditions change (e.g. changing input data volume, changing available resources, etc.), steps 3. - 5. can be repeated to optimize running workflows to the changed conditions.
7. **Retrieving and consuming results:**
Depending on the specification of the designed workflow, results of the streaming analysis workflow can be provided by different means. Output streams can be created, which can be further consumed by different streaming consumers (with or without using the INFORE Architecture). Snapshots of streamed results can be stored for further batch processing or inspection. Webservices or monitoring dashboards can deliver the results to the target users of the designed streaming analysis workflow.

This step-by-step process is also illustrated in Figure 29.

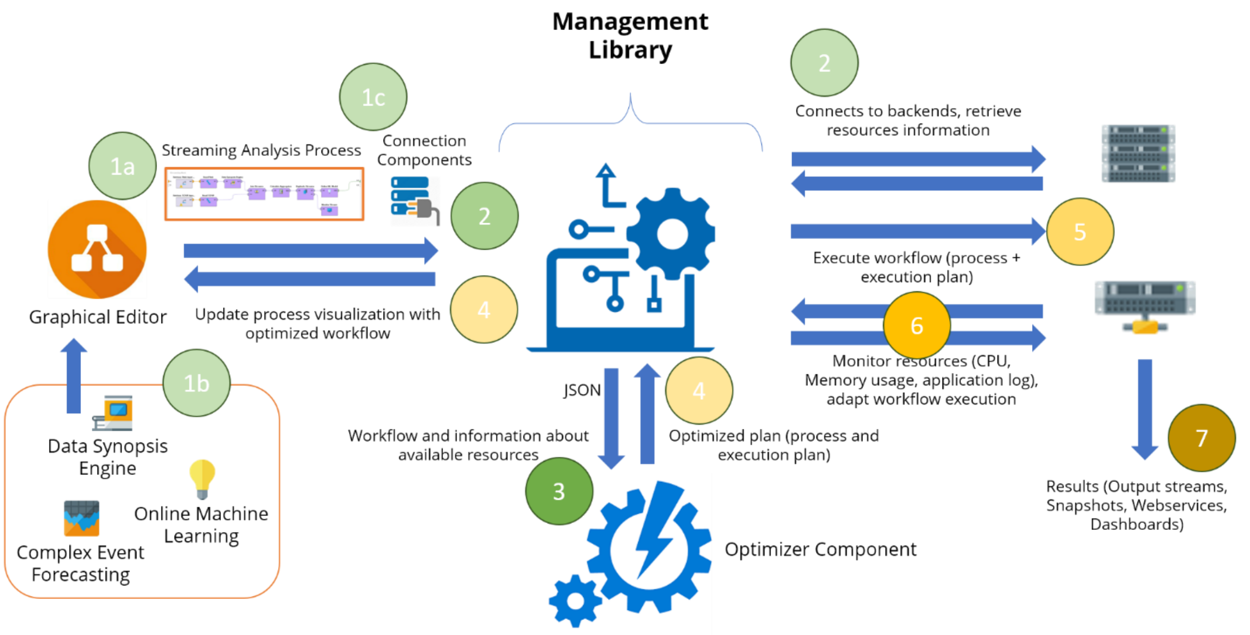

Management Library
1a
Streaming Analysis Process
1c
Connection Components
2
Connects to backends, retrieve resources information
Graphical Editor
Update process visualization with optimized workflow
4
5
Execute workflow (process + execution plan)
6
Monitor resources (CPU, Memory usage, application log), adapt workflow execution
1b
Data Synopsis Engine
Online Machine Learning
Complex Event Forecasting
JSON
Workflow and information about available resources
3
Optimized plan (process and execution plan)
Optimizer Component
7
Results (Output streams, Snapshots, Webservices, Dashboards)

Figure 29: Overview of the step-by-step process for creating and deploying a streaming analysis workflow in the INFORE Architecture.