



LarKC

*The Large Knowledge Collider
a platform for large scale integrated reasoning and Web-search*

FP7 – 215535

D11.4 Instrumentation and Monitoring platform - Realization

Coordinator: Ioan Toma (SG)

With contributions from: Ioan Toma (SG), Raluca Brehar (UTC), Silviu Bota (UTC), Mihai Chezan (SG), Ionel Giosan(UTC), Mihai Negru (UTC), Andrei Vatavu (UTC)

Quality Assessor: Vassil Momtchev (ONTO)

Quality Controller: Sergiu Nedevschi (UTC)

Document Identifier:	LarKC/2008/D11.4/V1.0
Class Deliverable:	LarKC EU-IST-2008-215535
Version:	version 1.0.0
Date:	July 29, 2011
State:	final
Distribution:	public

EXECUTIVE SUMMARY

This deliverable presents the final design and architecture of Semantic Instrumentation and Monitoring (SIM), the instrumentation and monitoring solution for LarKC plug-ins, workflows and platform components. It reports on the development and updates of each SIM component, namely *instrumentation mechanism, profiling agents, server, visualization* and *relevance feedback*, in terms of architecture and implementation. An updated installation and user guide is also provided in order to support interested users and developers to install and use SIM components, and also instrument LarKC plug-ins and workflows. As part of the final prototype of the instrumentation and monitoring solution, a large set of metrics, including methods, system, atomic and compound metrics is supported. We also reported on the set of workflows and their plugins that are instrumented and how visualization and relevance feedback components are using the monitoring data collected from these workflows and plugins.

DOCUMENT INFORMATION

IST Project Number	FP7 – 215535	Acronym	LarKC
Full Title	The Large Knowledge Collider: a platform for large scale integrated reasoning and Web-search		
Project URL	http://www.larkc.eu/		
Document URL			
EU Project Officer	Stefano Bertolo		

Deliverable	Number	11.4	Title	Instrumentation and Monitoring platform - Re-alization
Work Package	Number	11	Title	Instrumentation and Monitoring

Date of Delivery	Contractual	M40	Actual	31-July-11
Status	version 1.0.0		final	<input checked="" type="checkbox"/>
Nature	prototype <input checked="" type="checkbox"/> report <input type="checkbox"/> dissemination <input type="checkbox"/>			
Dissemination Level	public <input checked="" type="checkbox"/> consortium <input type="checkbox"/>			

Authors (Partner)	Softgress, UTC		
Resp. Author	Ioan Toma	E-mail	ioan.toma@softgress.com

Partner	Softgress, UTC	Phone	+40 7676 84924
----------------	----------------	--------------	----------------

Abstract (for dissemination)	This deliverable presents the final design and architecture of Semantic Instrumentation and Monitoring (SIM), the instrumentation and monitoring solution for LarKC plugins, workflows and platform components. It reports on the development and updates of each SIM component, namely <i>instrumentation mechanism, profiling agents, server, visualization and relevance feedback</i> , in terms of architecture and implementation. An updated installation and user guide is also provided in order to support interested users and developers to install and use SIM components, and also instrument LarKC plug-ins and workflows. As part of the final prototype of the instrumentation and monitoring solution, a large set of metrics, including methods, system, atomic and compound metrics is supported. We also reported on the set of workflows and their plugins that are instrumented and how visualization and relevance feedback components are using the monitoring data collected from these workflows and plugins.
Keywords	Instrumentation, Monitoring, Architecture

PROJECT CONSORTIUM INFORMATION

Participant's name	Partner	Contact
Semantic Technology Institute Innsbruck, Universitaet Innsbruck	 	Prof. Dr. Dieter Fensel Semantic Technology Institute (STI), Universitaet Innsbruck, Innsbruck, Austria Email: dieter.fensel@sti-innsbruck.at
AstraZeneca AB		Bosse Andersson AstraZeneca Lund, Sweden Email: bo.h.andersson@astrazeneca.com
CEFRIEL - SOCIETA CONSORTILE A RESPONSABILITA LIMITATA		Emanuele Della Valle CEFRIEL - SOCIETA CONSORTILE A RE- SPONSABILITA LIMITATA Milano, Italy Email: emanuele.dellavalle@cefriel.it
CYCORP, RAZISKOVANJE IN EKSPERI- MENTALNI RAZVOJ D.O.O.		Michael Witbrock CYCORP, RAZISKOVANJE IN EKSPERIMEN- TALNI RAZVOJ D.O.O., Ljubljana, Slovenia Email: witbrock@cyc.com
Höchstleistungsrechenzentrum, Universitaet Stuttgart		Georgina Gallizo Höchstleistungsrechenzentrum, Universitaet Stuttgart Stuttgart, Germany Email : gallizo@hlrs.de
MAX-PLANCK GESELLSCHAFT ZUR FOERDERUNG DER WISSENSCHAFTEN E.V.		Dr. Lael Schooler, Max-Planck-Institut für Bildungsforschung Berlin, Germany Email: schooler@mpib-berlin.mpg.de
Ontotext AD		Atanas Kiryakov, Ontotext Lab, Sofia, Bulgaria Email: naso@ontotext.com
SALTLUX INC.		Kono Kim SALTLUX INC Seoul, Korea Email: kono@saltlux.com
SIEMENS AKTIENGESELLSCHAFT		Dr. Volker Tresp SIEMENS AKTIENGESELLSCHAFT Muenchen, Germany Email: volker.tresp@siemens.com
THE UNIVERSITY OF SHEFFIELD		Prof. Dr. Hamish Cunningham, THE UNIVERSITY OF SHEFFIELD Sheffield, UK Email: h.cunningham@dcs.shef.ac.uk
VRIJE UNIVERSITEIT AMSTERDAM		Prof. Dr. Frank van Harmelen, VRIJE UNIVERSITEIT AMSTERDAM Amsterdam, Netherlands Email: Frank.van.Harmelen@cs.vu.nl
THE INTERNATIONAL WIC INSTI- TUTE, BEIJING UNIVERSITY OF TECHNOLOGY		Prof. Dr. Ning Zhong, THE INTERNATIONAL WIC INSTITUTE Mabeshi, Japan Email: zhong@maebashi-it.ac.jp
INTERNATIONAL AGENCY FOR RE- SEARCH ON CANCER		Dr. Paul Brennan, INTERNATIONAL AGENCY FOR RE- SEARCH ON CANCER Lyon, France Email: brennan@iarc.fr
INFORMATION RETRIEVAL FACILITY		Dr. John Tait, Dr. Paul Brennan, INFORMATION RETRIEVAL FACILITY Vienna, Austria Email: john.tait@ir-facility.org

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA http://www.utcluj.ro/		Prof. Dr. Eng. Sergiu Nedevschi TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Cluj-Napoca, Romania E-mail: sergiu.nedevschi@cs.utcluj.ro
SOFTGRESS S.R.L. http://www.softgress.com/		Dr. Ioan Toma SOFTGRESS S.R.L. Cluj-Napoca, Romania E-mail: ioan.toma@softgress.com

TABLE OF CONTENTS

LIST OF FIGURES	7
1 INTRODUCTION	8
2 SIM ARCHITECTURE AND COMPONENTS UPDATES	9
2.1 Instrumentation mechanism	9
2.1.1 SIM-Instrumentation internals	9
2.1.2 Information about collected metrics	14
2.1.3 Agent communication	16
2.2 Profiling Agents	16
2.3 Server	18
2.4 Relevance Feedback	20
2.5 Visualization	25
2.5.1 Visualization Components	25
2.5.2 Graph based visualization of monitoring ontology and data	31
3 INSTALLATION AND END-USER GUIDE	35
3.1 Instrumentation	35
3.1.1 How to enable SIM for LarKC	35
3.1.2 Understanding the results of instrumentation	36
3.1.3 Extending SIM	39
3.2 Visualization	40
3.3 Relevance Feedback	41
3.3.1 Compiling the application	41
3.3.2 Running an example	42
4 USING SIM TOOLS FOR INSTRUMENTING AND MONITORING LARKC PLUGINS AND WORKFLOWS	46
4.1 Instrumented Workflows	46
4.2 Visualization for instrumented workflows	47
4.3 Relevance Feedback for instrumented workflows	48
5 CONCLUSION	52

LIST OF FIGURES

2.1	SIM Instrumentation AspectJ Class Diagram	10
2.2	SIM-Instrumentation Data Collection Class Diagram	11
2.3	SIM Instrumentation LarKC AspectJ Class Diagram	12
2.4	SIM Instrumentation mechanism - Sequence Diagram	12
2.5	Profiling Agents - Class Diagram	17
2.6	Metrics data model	17
2.7	SIM Server Class Diagram	19
2.8	The RF architecture and modules interaction	22
2.9	The RF off-line training application - class diagram	23
2.10	The RF off-line training application - detailed class diagram	24
2.11	Visualization Architecture	25
2.12	MySQL Relational Database Schema	27
2.13	RRD access reading interface	29
2.14	Visualization Flowchart	30
2.15	Graph based visualization of monitoring ontology and data - Visualization of metrics ontology	31
2.16	Graph based visualization of monitoring ontology and data - Panel for metrics selection and metrics charts	32
2.17	Graph based visualization of monitoring ontology and data - Metrics charts	33
2.18	Graph based visualization of monitoring ontology and data - Metrics charts for selected intervals	33
2.19	Graph based visualization of monitoring ontology and data - Selecting the method metric type and method	34
3.1	Data loading in the off-line RF training application	42
3.2	Building and saving a RF machine learning model	43
3.3	Using the RF model for predicting the parameters of a query.	44
3.4	Loading and testing a Best configuration RF model	45
4.1	Plugins and queries corresponding to the LLD_REASONIG workflow	48
4.2	Query metrics	49
4.3	Metrics that are stored as time-series values. Portlet representation.	49

1. Introduction

The EU FP7 Large Knowledge Collider project (LarKC) is developing an infrastructure that supports large-scale reasoning over billions of structured data in heterogeneous data sets. Such an infrastructure ensures that computational components of that implement methods from diverse fields can be coherently integrated (e.g. retrieval and selection from IR, cognitive science or statistics; abstraction from machine learning and knowledge modeling; or decision methods from economics or decision theory), in order to coordinate large scale inference over distributed and heterogeneous information and resources. The infrastructure supports researchers and practitioners to run their own reasoning experiments and applications, and should allow for scaling well beyond what is currently possible. In this context, measuring and understanding how well the experiments are running becomes crucial. LarKC developers would like to know for their plug-ins and components how performant they are, how many resources they consume, how many times these plugins and components have been invoked and used, how much data they consume and produce, and so forth. Answers for all these questions, can be provided by what we call *instrumentation and monitoring*. WP11 is developing a set of tools that can be used to instrument and monitor LarKC specific applications.

This deliverable describes the final version of the *instrumentation and monitoring* solution developed in LarKC. It reports on the development and updates of each SIM component in terms of architecture and implementation. An updated installation and user guide is also provided in order to support interested users and developers to install and use SIM components, and also instrument LarKC plug-ins and workflows. The deliverable also reports on the set of workflows and their plugins that are instrumented and how visualization and relevance feedback components are using the monitoring data.

This deliverable is organized as follows. Chapter 2 describes the updates of the architecture and the latest developments in terms of implementation of each component of the architecture. Chapter 3 provides details on how the instrumentation and monitoring tools can be installed and used. Chapter 4 discusses the usage of SIM tools on concrete LarKC workflows and plugins that have been instrumented. Finally, Chapter 5 concludes the deliverable.

2. SIM Architecture and components updates

This chapter reports on the current updates of Semantic Instrumentation and Monitoring (SIM) and its components. As described in [1], SIM includes five major components i.e. *instrumentation mechanism, profiling agents, server, visualization and relevance feedback*. Since the last release, each SIM component has been further developed in order to support instrumentation and monitoring of a larger set of LarKC plugins and workflows based on an extended set of metrics. In the rest of the chapter we shortly revisit the functionality of each SIM component and we focus on describing the updates and extensions of each components since the initial release of SIM in M35.

2.1 Instrumentation mechanism

Instrumentation mechanism is responsible for inserting code that performs measurements in the key parts of the application that is instrumented. There are different ways in which instrumentation can be realized e.g. either by direct source code editing, by changing the already compiled sources (byte-code manipulation) or by following a mixed approach in which source code is marked, using annotations for example, to be instrumented by byte-code manipulation. The byte-code instrumentation can be done at compile time or at runtime.

SIM-Instrumentation uses AspectJ¹, a Java aspect oriented framework implementation. AspectJ allows SIM to inject instrumentation code into specific locations, using byte-code manipulation. The *aop.xml* file defines the aspects to apply on code at runtime and is read at startup by *aspectjweaver javaagent*. Basic measurements are obtained by instrumenting specific classes from LarKC. More precisely the following are instrumented: Larkc (platform startup), Executor (workflow creation), Sparql-Handler (query execution) and any implementation of Plugin (plugin execution). The following methods are targeted:

- eu.larkc.core.endpoint.sparql.SparqlHandler.handle;
- eu.larkc.core.executor.Executor.execute;
- eu.larkc.core.executor.Executor.getNextResults;
- eu.larkc.plugin.Plugin.invoke;

2.1.1 SIM-Instrumentation internals

Structure

SIM-instrumentation is structured into three main (sub)components/phases:

- code injection (done with the help of AspectJ constructs);
- metrics measurements and collection;
- agent communication;

Figure 2.1 contains the updated class diagram of the instrumentation mechanism.

Figure 2.2 contains the class diagram of the data collection mechanism.

¹<http://www.eclipse.org/aspectj>

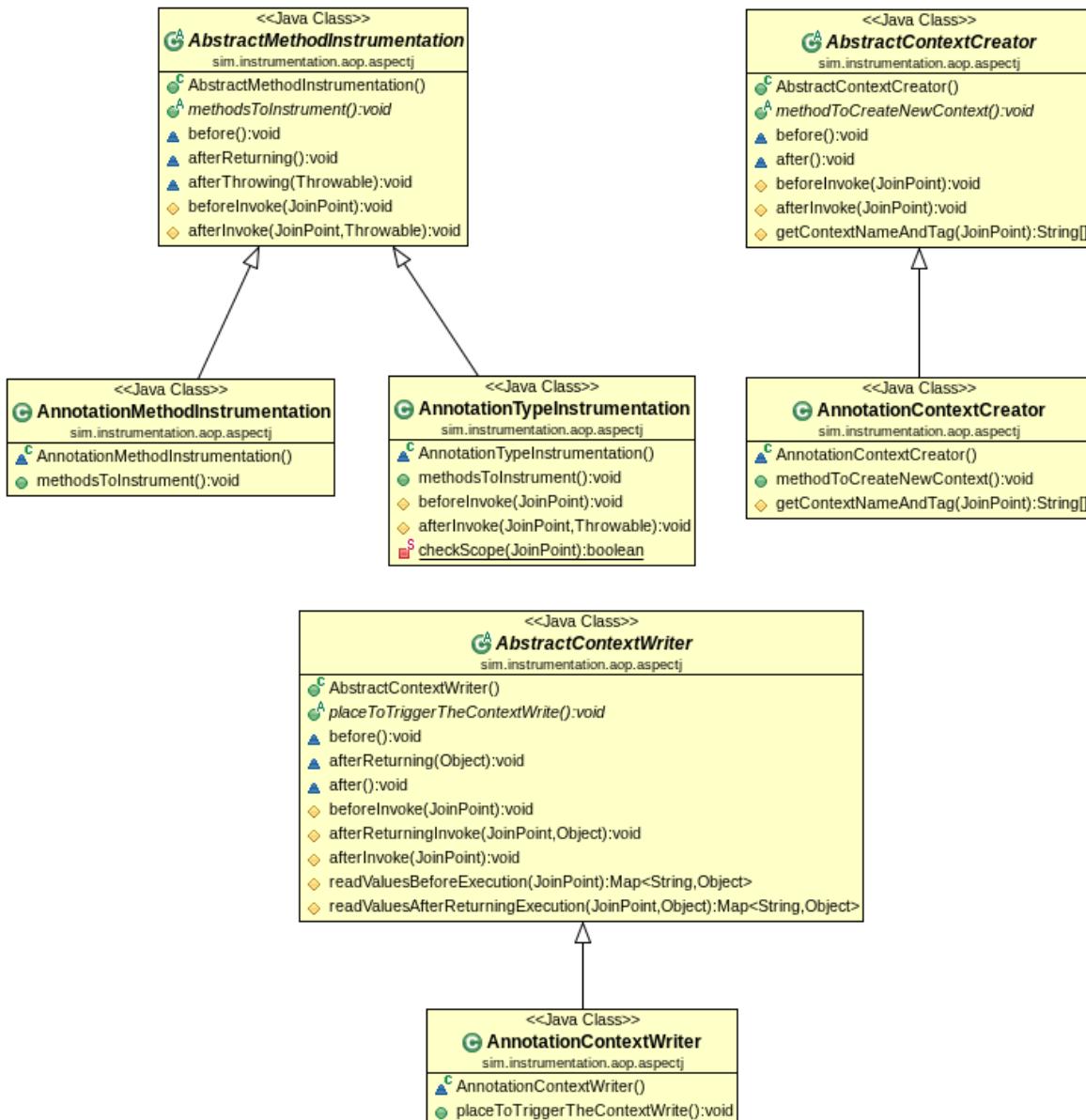


Figure 2.1: SIM Instrumentation AspectJ Class Diagram

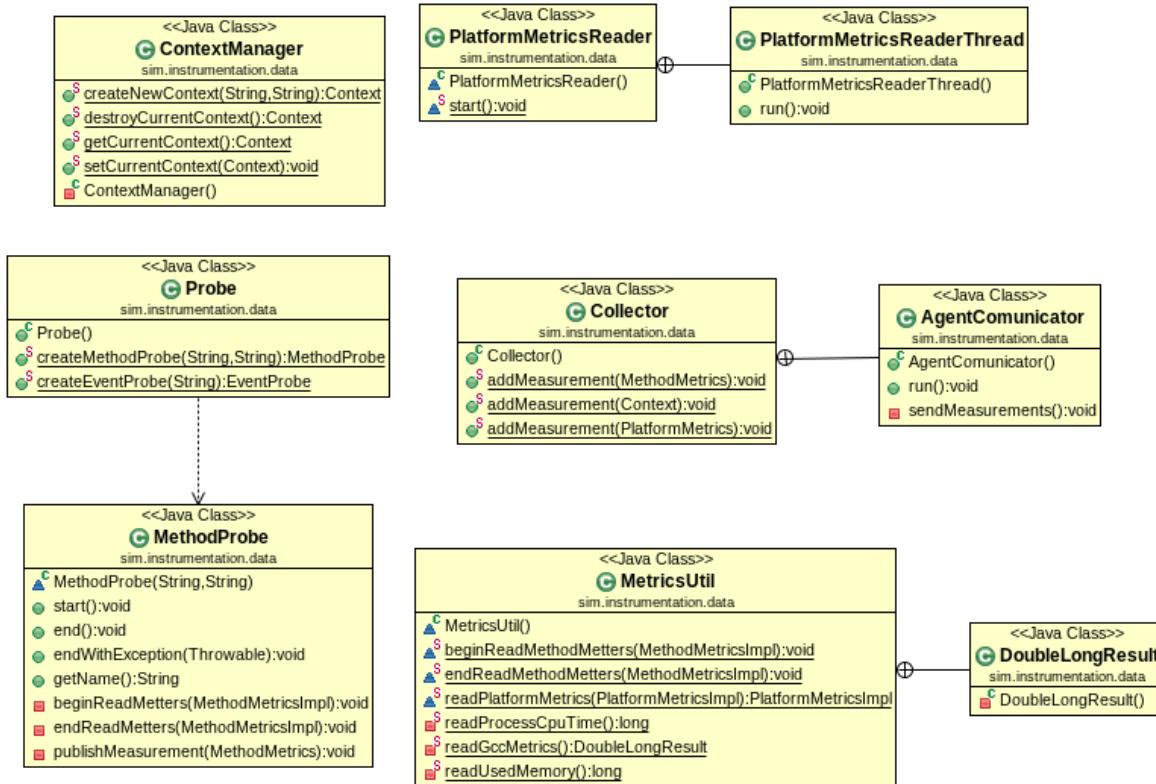


Figure 2.2: SIM-Instrumentation Data Collection Class Diagram

The updated class diagram for specific instrumentation of LarKC platform, plugins, workflows and queries is available in Figure 2.3.

The updated interaction between the (sub)components of the instrumentation mechanism is illustrated in Figure 2.4.

Project wise, SIM-instrumentation is split into:

1. *sim-instrumentation*, holding the generic instrumentation code, hosted @ <https://github.com/semantic-im/sim-instrumentation>
2. *sim-instrumentation-larkc*, holding the Larkc specific code, integrated into Larkc project

Code injection

sim.instrumentation.aop.aspectj.AbstractMethodInterceptor is the base aspect that handles all the AspectJ and metrics measurements plumbing. It defines the abstract pointcut *methodExecution()* which represents the place where the code injection will take place and where measurements will be performed. Concrete aspects extending *AbstractMethodInterceptor* need to implement this pointcut using standard AspectJ pointcut definition. For example to instrument all LarKC plugins one could define pointcut *methodToInstrument()* as:

```
public pointcut methodsToInstrument(): within(eu.larkc.plugin.Plugin)
&& execution(* *(..));
```

Another option is to use the *@Instrument* annotation. For example to instrument all Larkc plugins one needs to annotate *eu.larkc.plugin.Plugin* with *@Instrument* annotation.

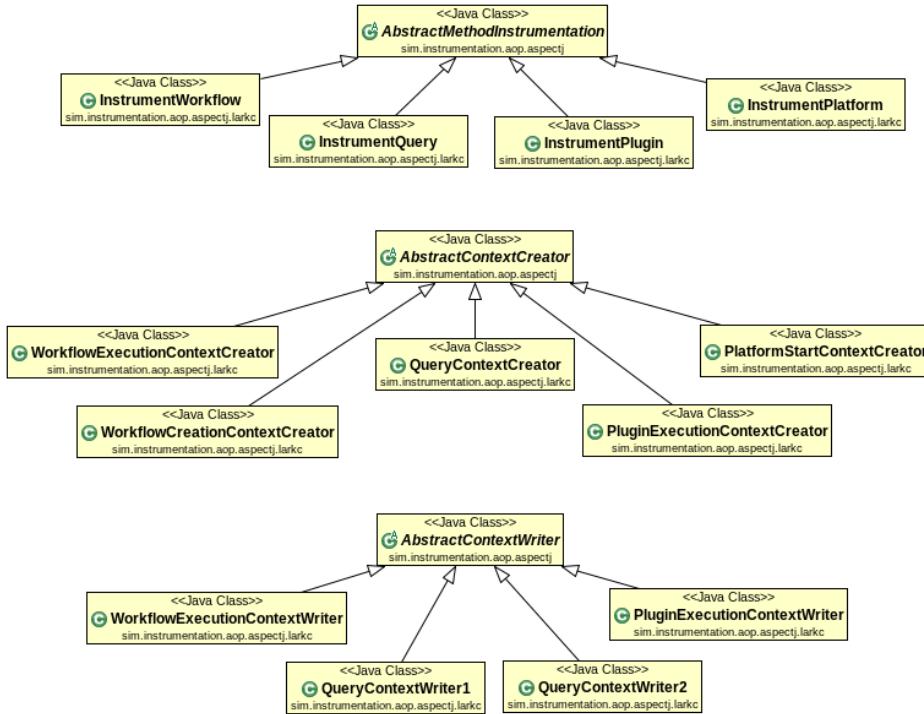


Figure 2.3: SIM Instrumentation LarKC AspectJ Class Diagram

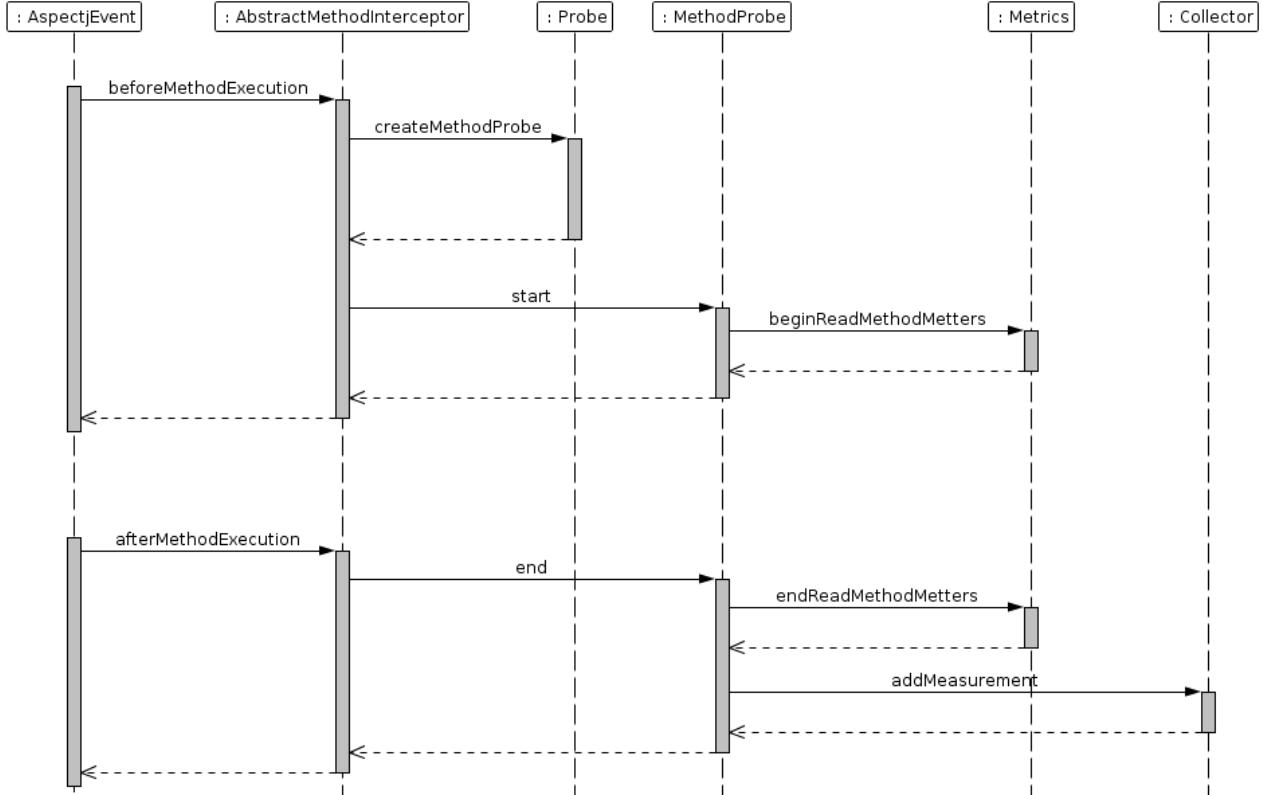


Figure 2.4: SIM Instrumentation mechanism - Sequence Diagram

```
@Instrument public abstract class Plugin \{\}\}
```

Context creation is handled by *AbstractContextCreator*, base abstract aspect used to indicate a method or constructor where a new context should be created for the current execution flow.

A *Context* is a container of information for an execution flow subgraph. It is uniquely identify by an id, it has a name and a tag and is linked to its parent *Context* through *parentContextId* property (which can be null in case this context is the root context). Any system can be seen as a black box that takes external input, processes it and produces the output. We call this process, of taking the input and producing the output, an "execution flow". So an execution flow can be seen as a series of operations, that can be executed either one after the other or in parallel or both. These operations can be grouped based on logical function they perform. Tracking these logical groups is made possible by creating one *Context* for each of them. Operations from one logical group will then use the *Context* of that logical group to publish information. For example, in case of an *ETL* (*Extract*, *Transform*, *Load*) type of execution flow, we would like to group the operations of this execution flow into the three logical functions, in our case: extract, transform, load. This can be done by creating three *Context* instances for each logical function. Operations executing for logical function *Extract* would operate on the *Extract Context*, those for *Transform* would operate on the *Transform Context* and those for *Load* would operate on the *Load Context*.

In order to create a new context for the current execution flow, one needs to define a concrete aspect of *AbstractContextCreator* that implements the *methodToCreateNewContext* pointcut, which defines the method or constructor that will create a new context when executed. Optionally, the concrete aspect can also override *getContextNameAndTag* method in order to define a custom name and tag for the new created context. Default implementation returns the class name of this joint point as the Name of the new context and the package name as the Tag.

An example on how to create a new context when method *myMethod* from *MyClass* is given below.

```
public aspect MyNewContext extends AbstractContextCreator {
    public pointcut methodToCreateNewContext(): execution(* MyClass.myMethod(..));
    protected String[] getContextNameAndTag(JoinPoint jp) {
        return new String[] {"CustomContextName", "CustomContextTag"};
    }
}
```

The same thing can be accomplished by using the *@CreateContext* annotation:

```
public class MyClass {
    @CreateContext(name="CustomContextName", tag="CustomContextTag")
    public void myMethod() {}}
```

Writing information into *Context* is done using *AbstractContextWriter*, base abstract aspect used to indicate a join point where values that are available at that join point should be written/published to the context of the current execution flow.

The *placeToTriggerTheContextWrite* pointcut defines the joint point where the reading of the values we want to publish to the context should happen. It can be a method or a constructor execution, a field get or set or an exception handler.

In order to publish values to the current context of the execution flow, one needs to define a concrete aspect of *AbstractContextWriter* that implements the *placeToTriggerTheContextWrite* pointcut. Optionally, the concrete aspect can also override *readValuesBefore* and/or *readValuesAfter* methods in order to publish custom values to the context. Default implementation will write to context the arguments (if there are any) in case the executing joint point is a method or constructor, the field value in

case of a field set and the exception message in case of exception. Also returns the return value (if there is any) in case the executing joint point is a method and the field value in case of a field get.

An example on how to publish values to the context when method myMethod from MyClass is given below.

```
public aspect MyContextWriter extends AbstractContextWriter {
    public pointcut placeToTriggerTheContextWrite(): execution(* MyClass.myMethod(..));
}
```

The same thing can be accomplished by using the `@WriteToContext` annotation:

```
public class MyClass {
    @WriteToContext
    public void myMethod() { ... }
}
```

2.1.2 Information about collected metrics

The generic part of SIM (sim-instrumentation) allows for the measuring of the following metrics on method invocation:

- wall clock time - elapsed time between method entry and method exit (ms)
- thread user cpu time - user CPU time spent by current thread executing this method (ms)
- thread system cpu time - system CPU time spent by current thread executing this method (ms)
- thread total cpu time - total CPU time spent by current thread executing this method (user time + system time) (ms)
- process total cpu time - total CPU time spent by current process (all threads from the application) while executing this method (ms)
- thread count - how many threads did this method invocation create (integer)
- thread block count - the total number of times that the current thread executing this method entered the BLOCKED state (integer)
- thread block time - the total accumulated time the current thread executing this method has been in the BLOCKED (ms)
- thread wait count - the number of times that the current thread executing this method has been in the WAITING or TIMED_WAITING state (integer)
- thread wait time - the total accumulated time the current thread executing this method has been in the WAITING or TIMED_WAITING state (ms)
- gcc count - total number of collections that have occurred while executing this method
- gcc time - approximate accumulated collection elapsed time in milliseconds while executing this method
- endedWithError - tells if the method ended with an uncaught exception

- exception - in case the method execution ended with an exception this is the exception.toString result

Every five seconds also generic platform measurement are collected:

- gcc count - total number of collections since platform start (count)
- gcc time - total accumulated collection elapsed time since platform start (ms)
- cpu time - total CPU time spent by current instrumented application since platform start (ms)
- uptime - total time since platform start (ms)
- average cpu usage - average cpu usage since platform start (%)
- cpu usage - cpu usage for the last 5 seconds (%)
- used memory - the amount of current used memory in bytes (bytes)

The LarKC specific part of SIM (sim-instrumentation-larkc) allows execution flow context information to be attached to the generic method invocation measurements. The information that is collected can be grouped in three major groups: query, workflow and plugin related information. They are specific method metrics measured on specific methods that correspond to query, plugin and workflow execution, metrics that are determine by processing the content of the query, or metrics that specify the size of data processed or produced by a plugin.

- **Query related metrics:** QueryContextInstance, QueryBeginExecutionTime, QueryEndExecutionTime, QueryErrorStatus, QueryContent, QuerySizeInCharacters, QueryNamespaceNb, QueryVariablesNb, QueryDataSetSourcesNb, QueryOperatorsNb, QueryResultOrderingNb, QueryResultLimitNb, QueryResultOffsetNb, QueryResultSizeInCharacters, QueryTotalResponseTime, QueryProcessTotalCPUTime, QueryThreadTotalCPUTime, QueryThreadUserCPUTime, QueryThreadSystemCPUTime, QueryThreadCount, QueryThreadBlockCount, QueryThreadBlockTime, QueryThreadWaitCount, QueryThreadWaitTime, QueryThreadGccCount, QueryThreadGccTime
- **Workflow related metrics:** WorkflowNumberOfPlugins, WorkflowTotalResponseTime, WorkflowProcessTotalCPUTime, WorkflowThreadTotalCPUTime, WorkflowThreadUserCPUTime, WorkflowThreadSystemCPUTime, WorkflowThreadCount, WorkflowThreadBlockCount, WorkflowThreadBlockTime, WorkflowThreadWaitCount, WorkflowThreadWaitTime, WorkflowThreadGccCount, WorkflowThreadGccTime
- **Plugin related metrics:** PluginName, PluginBeginExecutionTime, PluginEndExecutionTime, PluginErrorStatus, PluginTotalResponseTime, PluginProcessTotalCPUTime, PluginThreadTotalCPUTime, PluginThreadUserCPUTime, PluginThreadSystemCPUTime, PluginThreadCount, PluginThreadBlockCount, PluginThreadBlockTime, PluginThreadWaitCount, PluginThreadWaitTime, PluginThreadGccCount, PluginThreadGccTime, PluginInputSizeInTriples, PluginOutputSizeInTriples, PluginCacheHit

Generic metrics measurements such as *wallClockTime*, *threadUserCpuTime*, *threadCount*, *threadBlockTime*, *threadWaitTime*, *threadGccTime* are provided by the Util class *sim.instrumentation.data.Metrics*. In order to perform method metrics measurements *sim.instrumentation.data.Probe* builder should be used.

```
// before method invocation
MethodProbe mp = Probe.createMethodProbe(class_name, method_name);
mp.start()
// invoke method
...
// after method invocation
mp.end();
```

When *mp.end()* is called, metrics measurements is finalized and the measurements are also published to the *sim.instrumentation.data.Collector*, class responsible for collecting measurements and sending them to the agent.

2.1.3 Agent communication

The class *sim.instrumentation.data.Collector.AgentComunicator* implements the communication with profiling agents. *AgentComunicator* is running in a separate thread and sends every 5 seconds the collected metrics measurements to the agent using standard Java serialization over TCP sockets.

2.2 Profiling Agents

Collecting the monitoring data from the instrumentation mechanism and sending it to the server is the responsibility of profiling agents. A profiling agent is deployed on system where the instrumentation mechanism is running. The communication between the instrumentation mechanism and the agent is implemented in a client-server fashion. The instrumentation mechanism plays the role of an active client which sends monitored data from the application being instrumented to the agent. The agent can be seen as a lightweight server that collects data received from the instrumentation mechanism. Furthermore, the agent collects generic system measurements for the node where it is deployed (e.g. total memory or CPU consumption of the node).

Figure 2.5 contains the updated class diagram of the profiling agents.

The agent received metrics measurements from the instrumentation mechanism at the regular intervals. The updated internal data model used for representing metrics measurements is available in Figure 2.6.

The same data model is used for further communication between the agent and the server component. To enable easy access to generic system measurements, profiling agents expose such data through Java Management Extensions (JMX)² interface. In this way any tool, for example graphical tool, that "knows" JMX can connect to the agent, receive and display generic system measurements. The agents also communicate with the server, the central point where the data is stored and aggregated. The communication approach is implemented in the same client-server approach as between instrumentation mechanism and the agent. Other features that are supported include: (i) buffering of data received from the instrumentation mechanism and (ii) reliable communication between the agent and server e.g. if agent terminates, for various reasons, and is not able to send the entire data to the server, when restarted it must be able to send the rest of the data.

²<http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html>

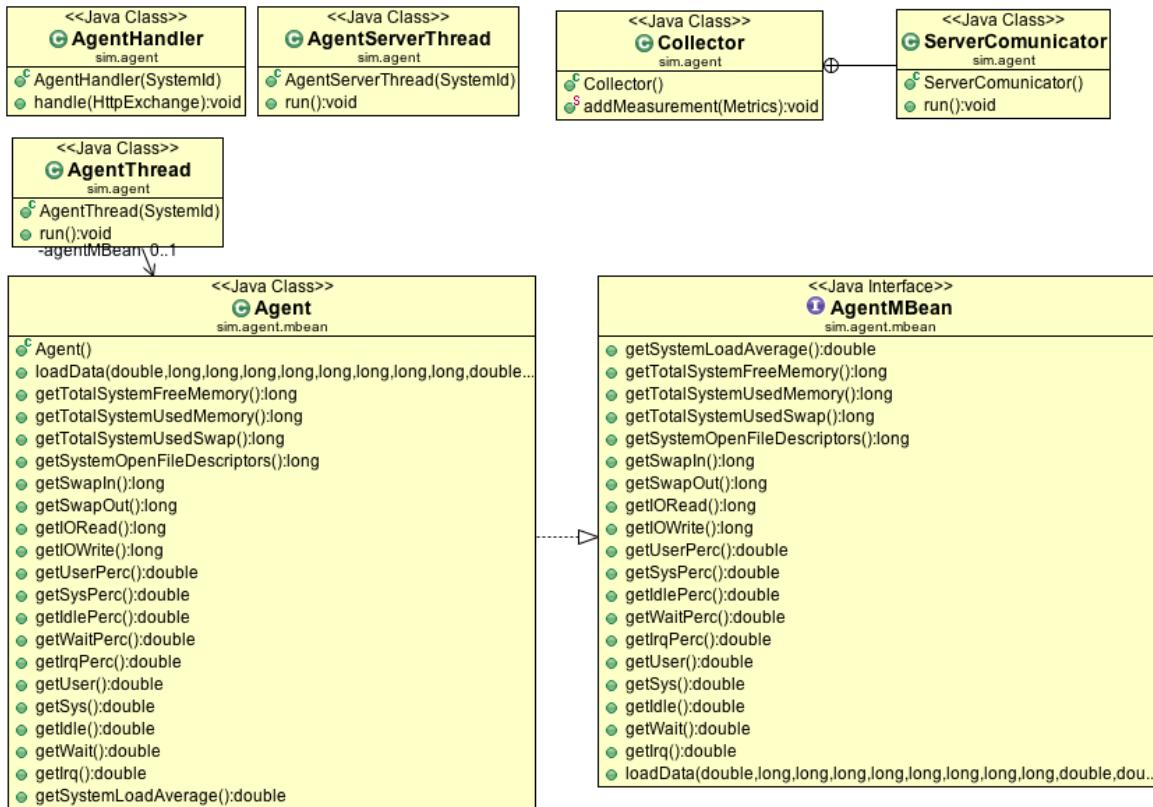


Figure 2.5: Profiling Agents - Class Diagram

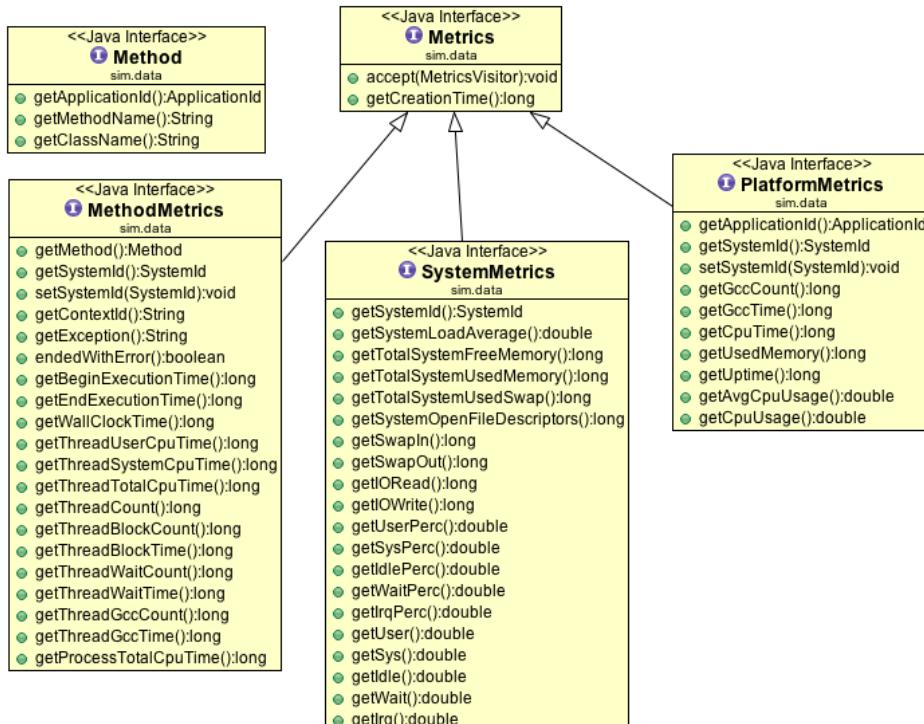


Figure 2.6: Metrics data model

2.3 Server

The server component stores collected monitoring data using three technical solutions, namely: (i) Round Robin Database (RRD), (ii) Relational Database (RDBMS) and (iii) Semantic storage.

The semantic storage component retrieves monitoring data from the profiling agents, creates RDF statements according to the metrics ontology described in [2], and stores these statements in the underlying RDF storage. The RDF2Go³ library is used to communicate with the underlying RDF storage. RDF2Go is an API that abstracts from the underlying RDF storage and provides connectors for various RDF storages such as OWLIM or Sesame. Using RDF2Go, the SIM server communicates with OWLIM⁴ or Sesame⁵. Data from the semantic storage component is used by the relevance feedback and visualization component. These components access the data by executing SPARQL queries against the semantic storage. The retrieved data, in RDF format, is then transformed in the format required by these components (e.g. ARFF format in the case of relevance feedback).

Round Robin Database (RRD) technology is used for its feature of aggregating series of measurements of metrics such as CPU consumption, memory usage, etc. The values of these metrics are changing frequently (e.g. frequent change of CPU consumption of a node where a plug-in is running). RRD is not used as overall storage for all measurements but rather as a support tool for aggregation of numerical measurements.

RDBMS are used as an alternative solution for storing the monitoring data being well known, established and familiar technology. The visualization component in particular uses the RDBMS storing solution. The semantic and RDBMS-based storage mechanisms are used as overall storage mechanisms. SIM users can choose to have the overall storage solution based on semantic or RDBMS repositories.

More details on how relevance feedback and visualization components interact with the storage mechanisms, available as part of the server component, are provided in Section 2.4 and Section 2.5.

The updated class diagram of the server component is available in Figure 2.7.

Besides storing monitoring data, the server component is also capable of aggregating it into more complex metrics, called compound metrics. The compound metrics are computed periodically (every hour and every day) based on atomic metrics values. To retrieve the values of atomic metrics stored in the data layer, SPARQL queries are executed at fix time intervals. Once retrieved they are aggregated into compound metrics which are written back into the data layer. The following compound metrics are currently supported by SIM Server component.

- QueriesPerTimeInterval - counts the number of queries that were received in a given time interval
- QuerySuccessRatePerTimeInterval - counts the number of queries that finished with success
- QueryFailureRatePerTimeInterval - counts the number of queries that finished with failure

³<http://semanticweb.org/wiki/RDF2Go>

⁴<http://www.ontotext.com/owlim/>

⁵www.openrdf.org/

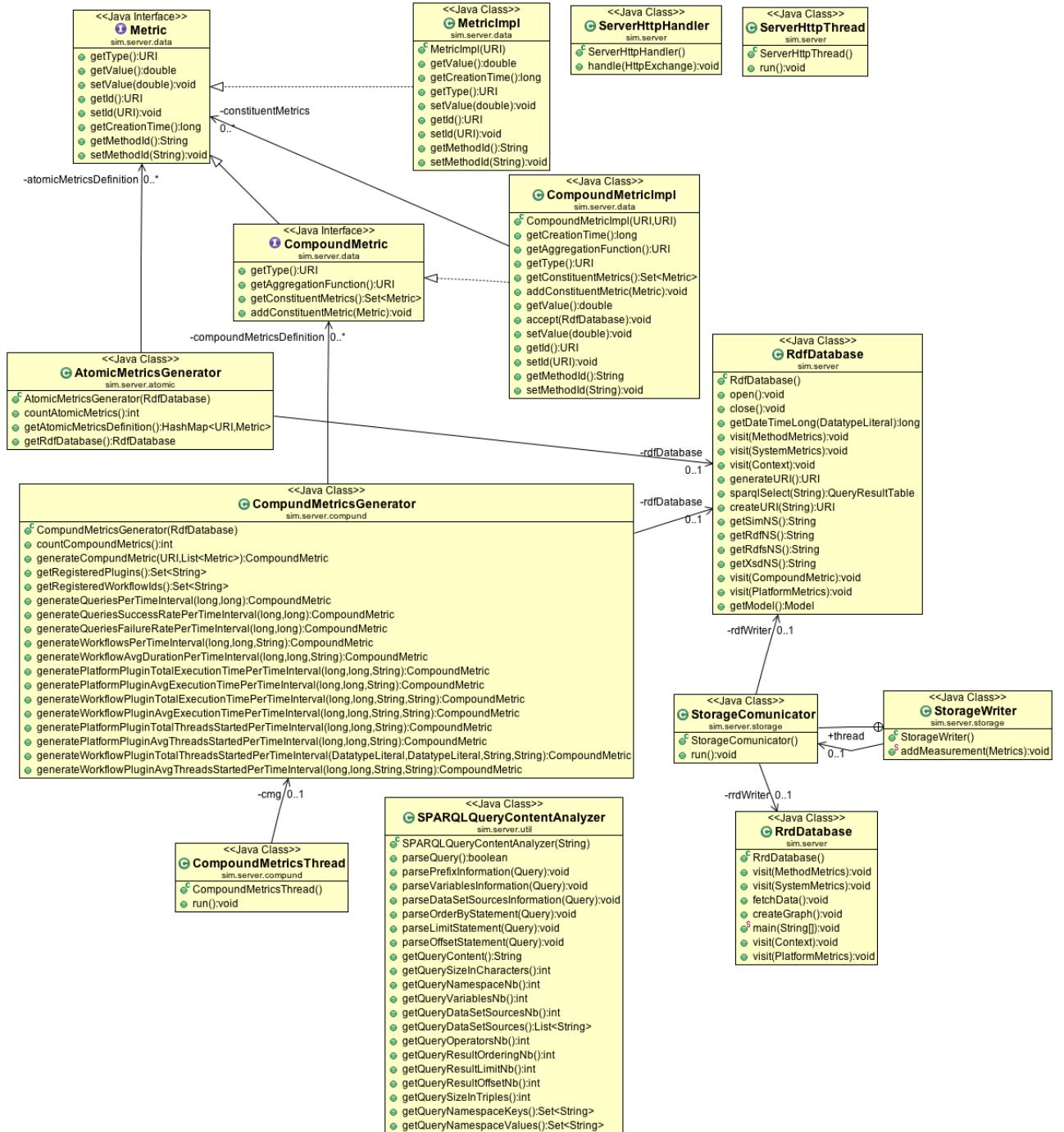


Figure 2.7: SIM Server Class Diagram

- WorkflowsPerTimeInterval - counts the number of workflows of a given name (id) that were started in a given time interval
- WorkflowAvgDurationPerTimeInterval - the average duration of a given workflow in a given time interval
- PlatformPluginTotalExecutionTimePerTimeInterval - total execution time of the given plug-in in a given time interval.
- PlatformPluginAvgExecutionTimePerTimeInterval - average execution time of the given plug-in in a given time interval.
- WorkflowPluginTotalExecutionTimePerTimeInterval - total execution time of the given plug-in in a given time interval for the given workflow.
- WorkflowPluginAvgExecutionTimePerTimeInterval - average execution time of the given plug-in in a given time interval for the given work-flow.
- PlatformPluginTotalThreadsStartedPerTimeInterval - total number of threads of the given plug-in in a given time interval.
- PlatformPluginAvgThreadsStartedPerTimeInterval - average number of threads started by the given plug-in in a given time interval.
- WorkflowPluginTotalThreadsStartedPerTimeInterval - total number of threads started by the given plug-in in all the runs of a specified workflow in a given time interval.
- WorkflowPluginAvgThreadsStartedPerTimeInterval - average number of threads of the given plug-in in all the runs of a specified workflow in a given time interval.

2.4 Relevance Feedback

The relevance feedback component comprises the on-line application module and the off-line training module:

- the off-line training module - consists in several data mining algorithms. It has the role of building prediction models that based on previously recorded values for input and output metrics is able to find the relationship between input and output metrics.
- the on-line application module - applies the model defined by the off-line training module, such that given the input metrics it returns the values of output metrics (expected values) and feedback based on the function defined in the prediction models.

The relevance feedback top-level architecture is depicted in Figure 2.8. The remark is that the whole architecture interacts directly or indirectly with three other entities: the user, the visualization module and the data storage container.

In the on-line part, the user interacts directly with the visualization component by sending the input query, workflow and constraints that are subject of the prediction process. The visualization module sends data to the model loader which should select the proper prediction model used for building the prediction and feedback results. The

interaction with the data storage container refers to the fact that the model loader takes the proper prediction model by querying the models' database. The chosen prediction model is sent back to the loader which sends it further to the applying prediction component. In this part, the new query, workflow and constraints both with the prediction model lead to some output predicted metrics values and feedback. These results are sent back to the visualization module and the user can analyze them. The whole on-line process should be fast enough to make a reasonable prediction such that the user does not wait a long period of time.

The off-line part is implemented in a standalone application which is described in more details in the “End-user guide” chapter. The data collector module connects to the data storage component, gets the values of input and output metrics needed for training the prediction models and transform them in Weka specific ARFF files (see deliverable D11.2). The data preprocessing module gets the initial raw data set from the ARFF files and applies some transformations to obtain the selected features data set compatible with the next prediction model builders. Then, a set of prediction models are built to satisfy all the scenarios proposed (D11.3, chapter 4): scalability analysis, bottleneck prediction, workflow prediction based on raw data. All the prediction modules take the selected features data as the input and offer the corresponding prediction models as the result. We have implemented the following prediction models corresponding to each model builder from the architecture schema:

- Clustering&Regression (D11.3, section 3.3) which satisfies both the scalability analysis and bottleneck prediction scenarios.
- Kernel-Regression which estimates the conditional expectation of a random variable, so that it predicts one output metric conditioned by the set of corresponding input values. It satisfies the scalability analysis prediction scenario.
- Best configuration (D11.3, section 4.3) which selects the best workflow that may solve a given query and predicts the output metrics for that situation. It satisfies the work-flow prediction based on raw data scenario.
- Kernel Canonical Correlation Analysis (D11.3, section 3.4) is used for finding a relationship between input and output metrics and predict the output metrics values knowing the input values and the previously computed correspondence. It solves the scalability analysis scenario.

All prediction models are saved into the data storage container by the model saver module for later use in the on-line predictions and feedback. The data storage is used by the relevance feedback component both in on-line and off-line processes.

The updated overall design of the off-line relevance feedback application is presented in 2.9. Although, many of the classes were described in D11.2 section 2.2.5 we present a short description of all classes and packages:

- Weka machine learning library is used for defining some specific data structures (ARFF files and instances) and for solving some particular operations.
- Jena library package is used for parsing the new input SPARQL queries.
- Jama library offers support for matrix operations which is very useful in Kernel-Regression module.
- The “MainFrame” class contains all the design and associated functionalities of the whole off-line application user interface.

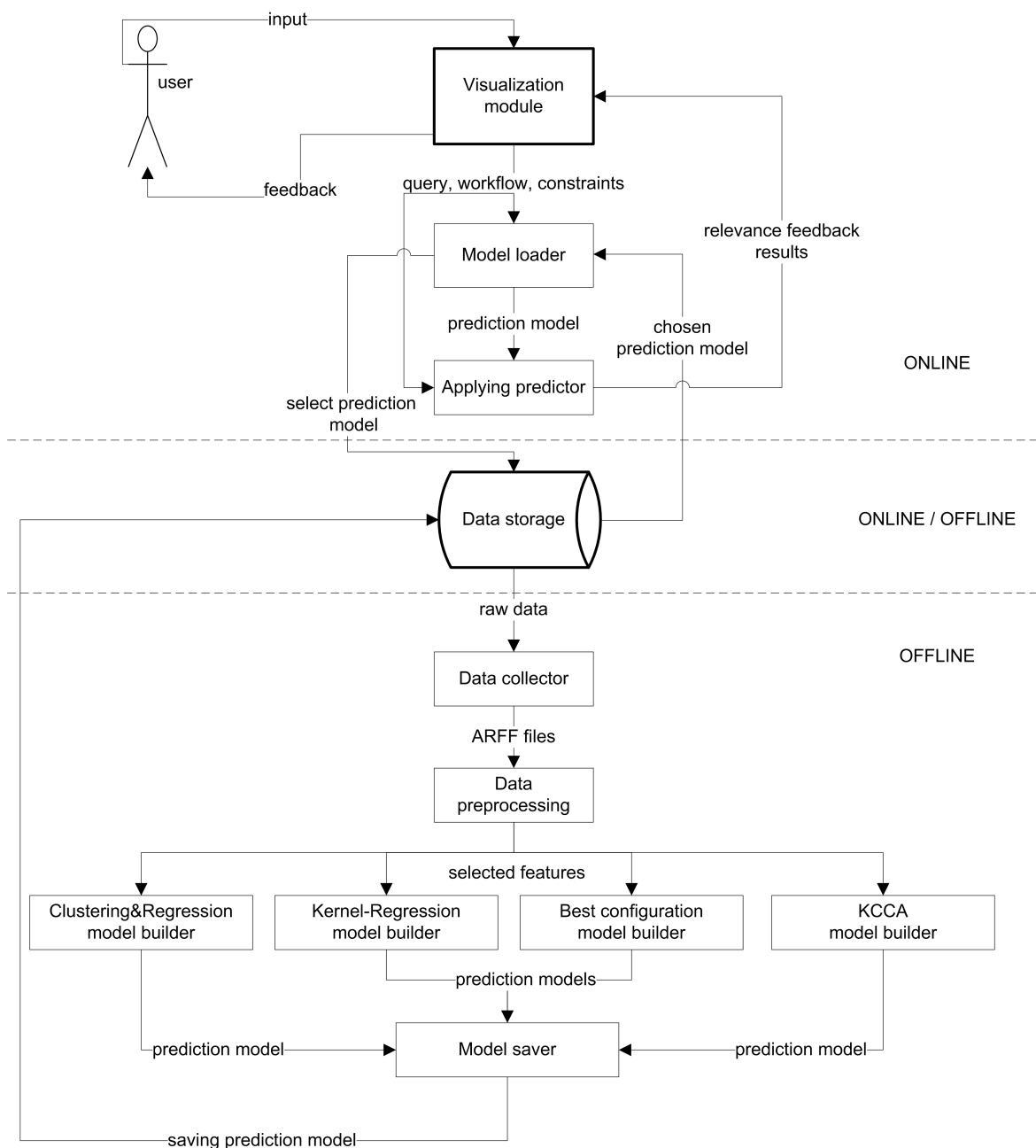


Figure 2.8: The RF architecture and modules interaction

- The “FileFilterExt” class is used for the GUI interaction with the ARFF data files (loading, saving).
- The “DataPreprocessor” gets an initial raw data set and apply a set of transformations to obtain a compatible data set for the prediction classes.
- The “ClusterMain” class has the responsibility to apply a expectation-maximization clustering technique on the training data.
- The “Predictor” class is a generic predictor used in computing all the prediction models that will be used.
- The “RFClusteringRegression”, “KernelRegression”, “BestConfigurator” classes implement the prediction functionalities, stated in the previous paragraphs, in correspondence with the proposed scenarios. They also have functionalities for saving/loading models to/from data storage.

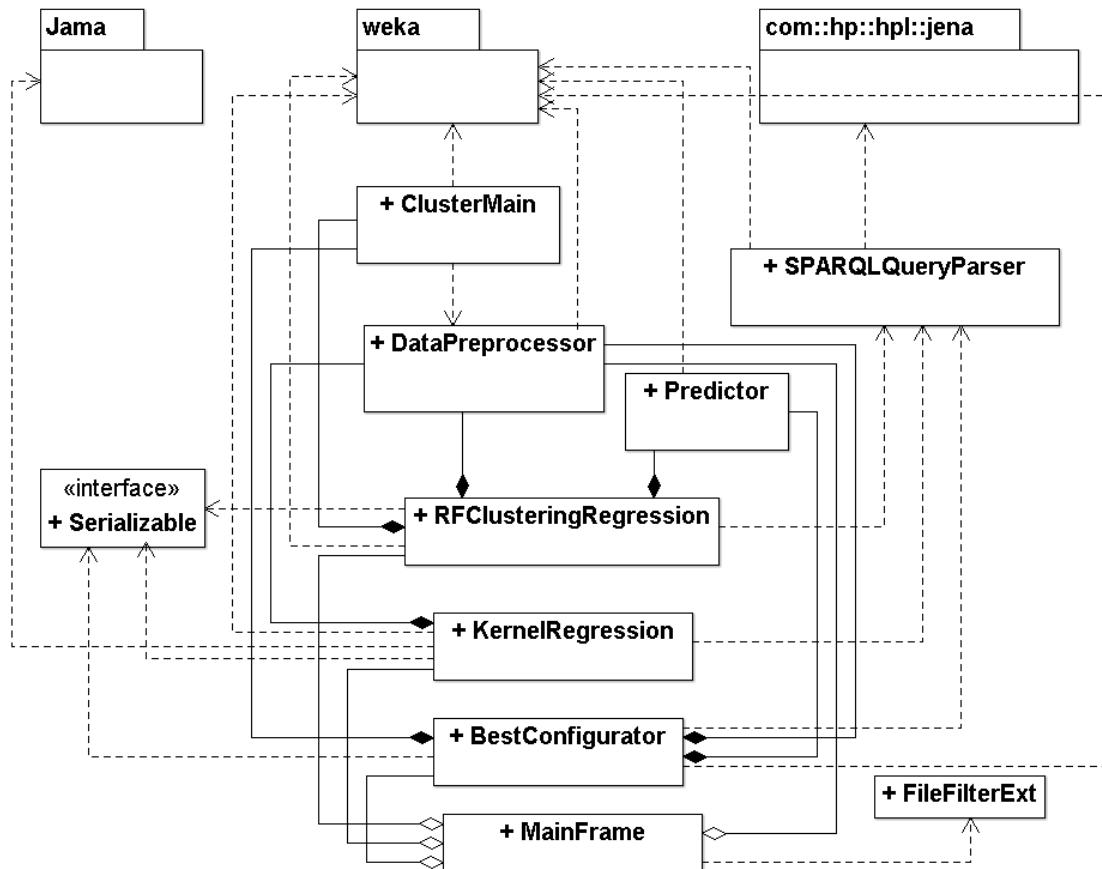


Figure 2.9: The RF off-line training application - class diagram

The detailed class-diagram with all class attributes, methods and relationships of the training system is presented in Figure 2.10.

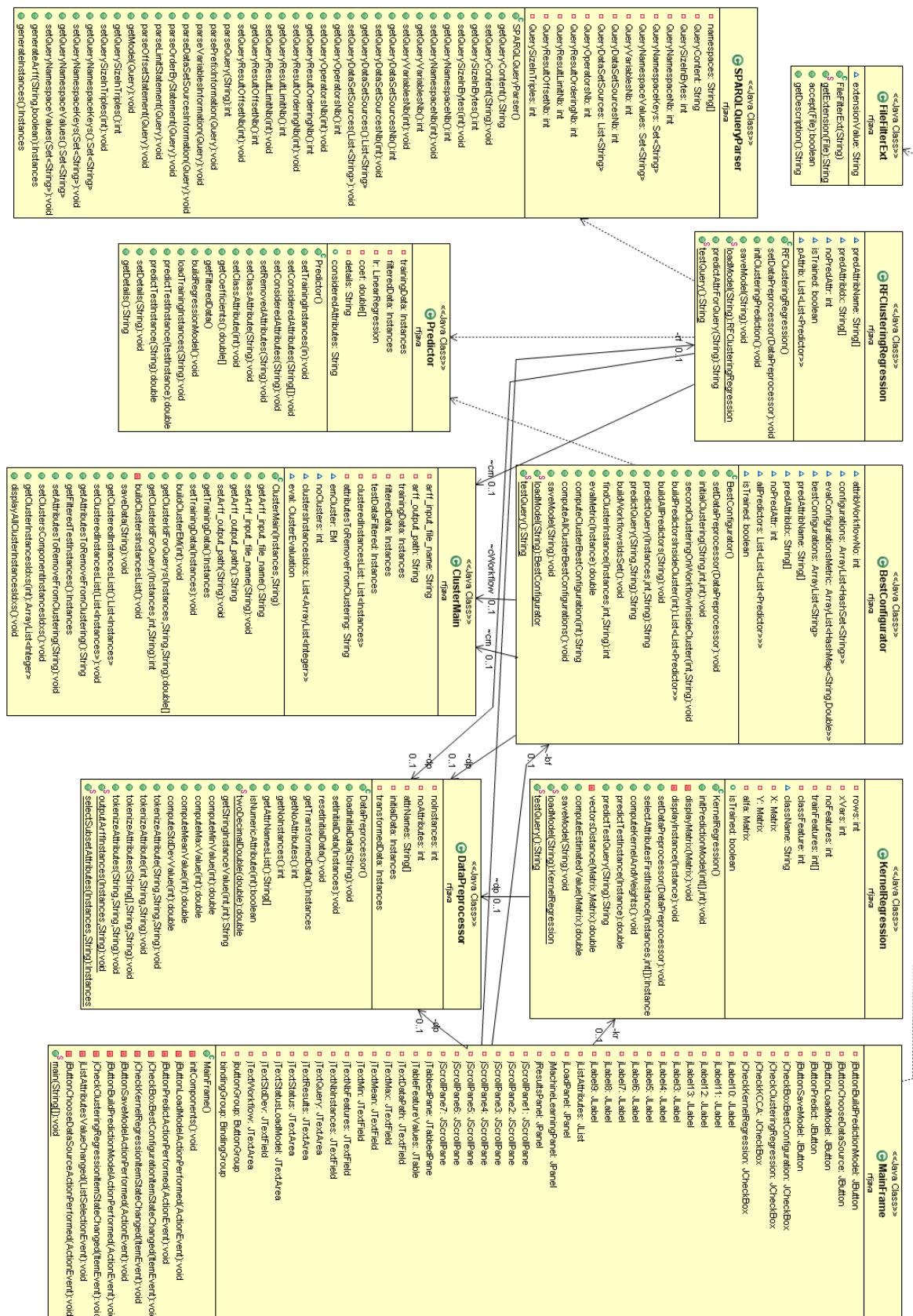


Figure 2.10: The RF off-line training application - detailed class diagram

2.5 Visualization

The visualization, analysis and reporting tool allows us to enable easy interpretation of the data collected during instrumentation and monitoring. Using this tool, LarKC users and developers can visualize real-time or historical data. Furthermore they can analyze reports based on aggregated metric values. The Visualization module does not interact directly to the LarKC platform. The instrumentation agents are grabbing and writing metrics data in the visualization data layer. The metrics are transformed then by the visualization logic into a more intuitive representation at the platform, workflows, plugins and queries levels.

Figure 2.11 describes the general architecture of the visualization framework.

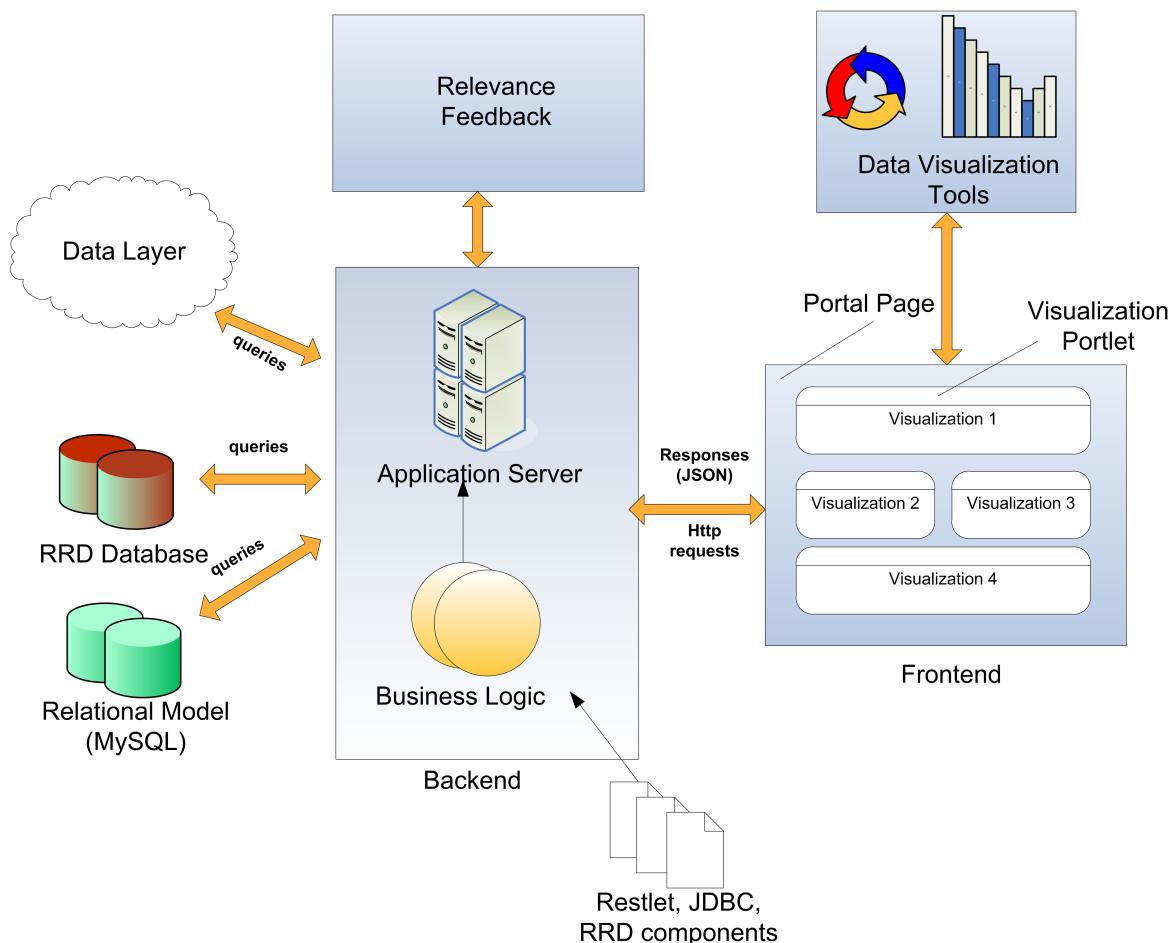


Figure 2.11: Visualization Architecture

2.5.1 Visualization Components

Our visualization module is responsible for (1) displaying data and metrics about the LarKC platform, plug-ins and workflows and (2) for sending input metrics to the relevance feedback module and displaying the results provided by the relevance feedback. The visualization is a web-based, client-server application with several modules for

representing real-time and historical data, composed of atomic and compound metrics about:

- LarKC platform in general.
- Queries that were passed to the LarKC platform.
- Workflows used to execute the queries.
- Plug-ins that compose the workflows.

The visualization module is separated into the following main components: Client-Side Component, Server Side Business Logic and Data Layer Component. The description of these components and concepts for best practices can be found in deliverable [3].

Traditional web applications contain several HTML pages and links between those pages. Usually, for the client side implementation Google Web Toolkit (GWT) works with a single HTML page. This could be a problem when an application includes more pages, because GWT was meant to be used as a standalone web application development platform. Another issue is that GWT does not allow loading multiple modules into a single page and does not integrate well with existing HTML elements.

To overcome this problems one of the solutions is to adopt the JSR-286 portlet standard. Thus a visualization scenario can be split into many atomic tasks. Each visualization task can be considered as an atomic portlet (widget) that could be aggregated at a higher level into a single web-based environment.

In order to provide a scalable and easy to maintain solution, for the server side business logic, we chose the Liferay Portal as the web application aggregator and content management system. Liferay is an open source portal platform that allows users to create relevant content, and integrate their custom functionality into a single web-based environment. The Liferay interface is composed of smaller, self-contained web elements, called Portlets. The portlet specification is included in JSR-168 standard. Liferay also supports Inter Portlet Communication (IPC), introduced in JSR-286. Therefore, the visualization component philosophy consists in developing new visualization portlets and adapting the existing ones to the portlet specification and aggregating them into a single visualization portal environment according to existing requirements.

Some of the features offered by Liferay are listed below:

- Liferay provides Content Management Functionalities in a single portal environment.
- It's easy to use, extensible, integrated with other tools and standards.
- It's free and open source (LGPL License).
- User interface development and customization can be made by using the Liferay's plugins SDK.
- Each visualization task can be developed as separate portlet project by different developers and integrated subsequently into the visualization platform.

Two mechanisms for client-server communication can be used: by making GWT Remote Procedures Calls or by retrieving JSON Data via HTTP requests (see [3]).

For data persistence the visualization interacts with two kinds of databases:

- Relational DB: MySQL database - permits to create a consistent, logical representation of metrics and other relevant features. The MySQL queries and responses are coordinated by the server through the JDBC connector.
- RRD database: able to store time-series values. In order to handle this type of data we use the RRDTool component.

MySQL database

The relational data base schema we propose is described in Figure 2.12.

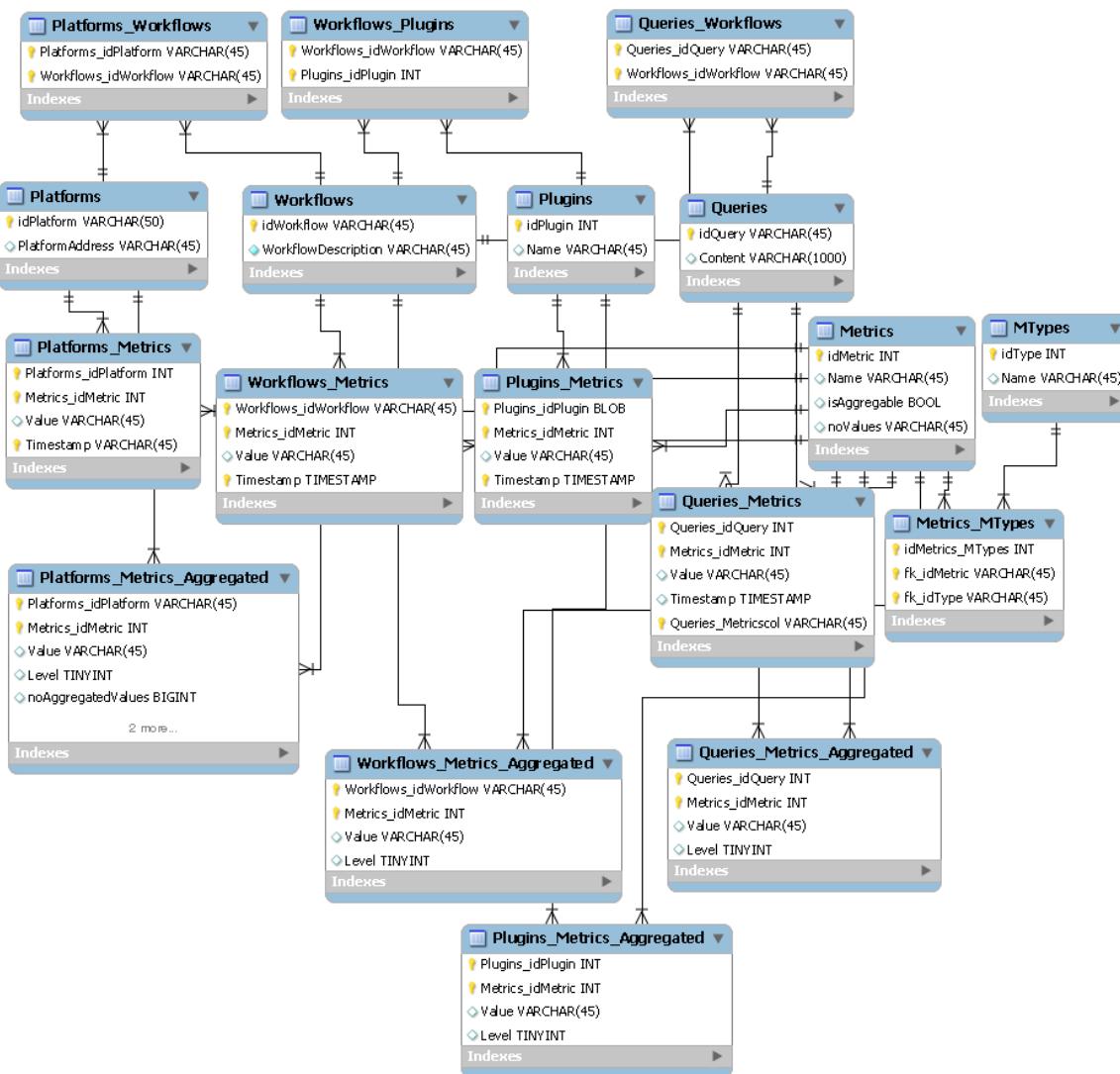


Figure 2.12: MySQL Relational Database Schema

In the defined model, the “Platforms” table stores data about each instance of the platform that has been instrumented (a unique identifier is provided to each platform). Each platform has a list of workflows that have been run by it and the connection between the platform and the workflows is done via the table “Platform_Workflows”.

Each workflow can be run for a query (also uniquely identified) or more queries and the linking is made by “Queries_Workflows” table.

The relation between the list of plug-ins and a workflow for which they are run is done via the table “Workflows_Plugins”. The description of plugins is stored in table “Plugins” and the description of workflows is stored in table “Workflows”.

The information about metrics is collected in table “Metrics” that lists the names of all the metrics and “MTypes” that contains the types of metrics (like atomic, compound, query metric, workflow metric, plug-in metric, etc). The values of the metrics are stored in the tables: “Platforms_metrics”, “Workflows_Metrics”, “Plugins_Metrics”, “Queries_Metrics”.

RRD database

An important aspect of monitoring is the ability to view real time data, such as the processor load, memory and other resource utilization etc. This type of data can best be represented as a time series. We cannot, however, continuously store such data, because the storage size will continually increase. Therefore, only relatively recent data should be stored. Old data should be aggregated. The de facto standard for high performance storage and access to such data is RRDTool⁶.

An RRD database (which in the usual DBMS parlance would be best described as just a table) contains one or more data series, but all of them must be sampled at the same time points (i.e. there is a single time stamp column, and any number of value columns.) The number of the entries in the table is limited to a configurable value. Besides the values themselves, the database can also contain aggregated (archived) data. Archived data is obtained by applying an aggregation function (such as AVERAGE, MAX, MIN) on accumulated data. The number of aggregated entries is also fixed and configurable.

Because all the columns must share the same time stamp, and we chose to use multiple databases, each database containing a single data column, corresponding to one atomic metric. Currently, we have the following databases: *IdlePercent*, *IdleTime*, *IOIn*, *IOOut*, *IRQPercent*, *IRQTime*, *SwapIn*, *SwapOut*, *SysPercent*, *SystemLoadAverage*, *SystemOpenFileDescriptorCount*, *SysTime*, *TotalSystemFreeMemory*, *TotalSystemUsedMemory*, *TotalSystemUsedSwapSpace*, *UserPercent*, *UserTime*, *WaitPercent*, *WaitTime*. More databases can be created as needed.

RRDTool is a RRD application written in the C programming language. In order to use RRD from Java we needed a Java based implementation. Currently there are two such implementation and they are almost equivalent. The first one is RRD4J⁷ distributed under the Apache 2.0 license and the second one is JRobin⁸ distributed under the LGPL license. Because they are similar we provided interfaces for both of them.

Our interface allows both writing data into the RRD and reading data from it. Writing data is straight-forward; it is embedded directly in the server code. Data reading (publishing) is done using a RESTlet component⁹. The data is transmitted using the JavaScript Object Notation (JSON¹⁰). The object is an array, containing time stamp – value pairs.

⁶<http://www.mrtg.org/rrdtool>

⁷<https://rrd4j.dev.java.net/>

⁸http://www.jrobin.org/index.php/Main_Page

⁹<http://www.restlet.org/>

¹⁰<http://www.json.org/>

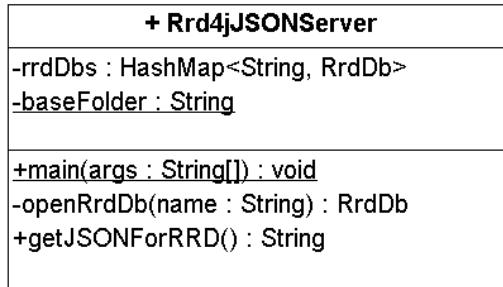


Figure 2.13: RRD access reading interface

The interaction between the main visualization modules and the users is presented in figure 2.14.

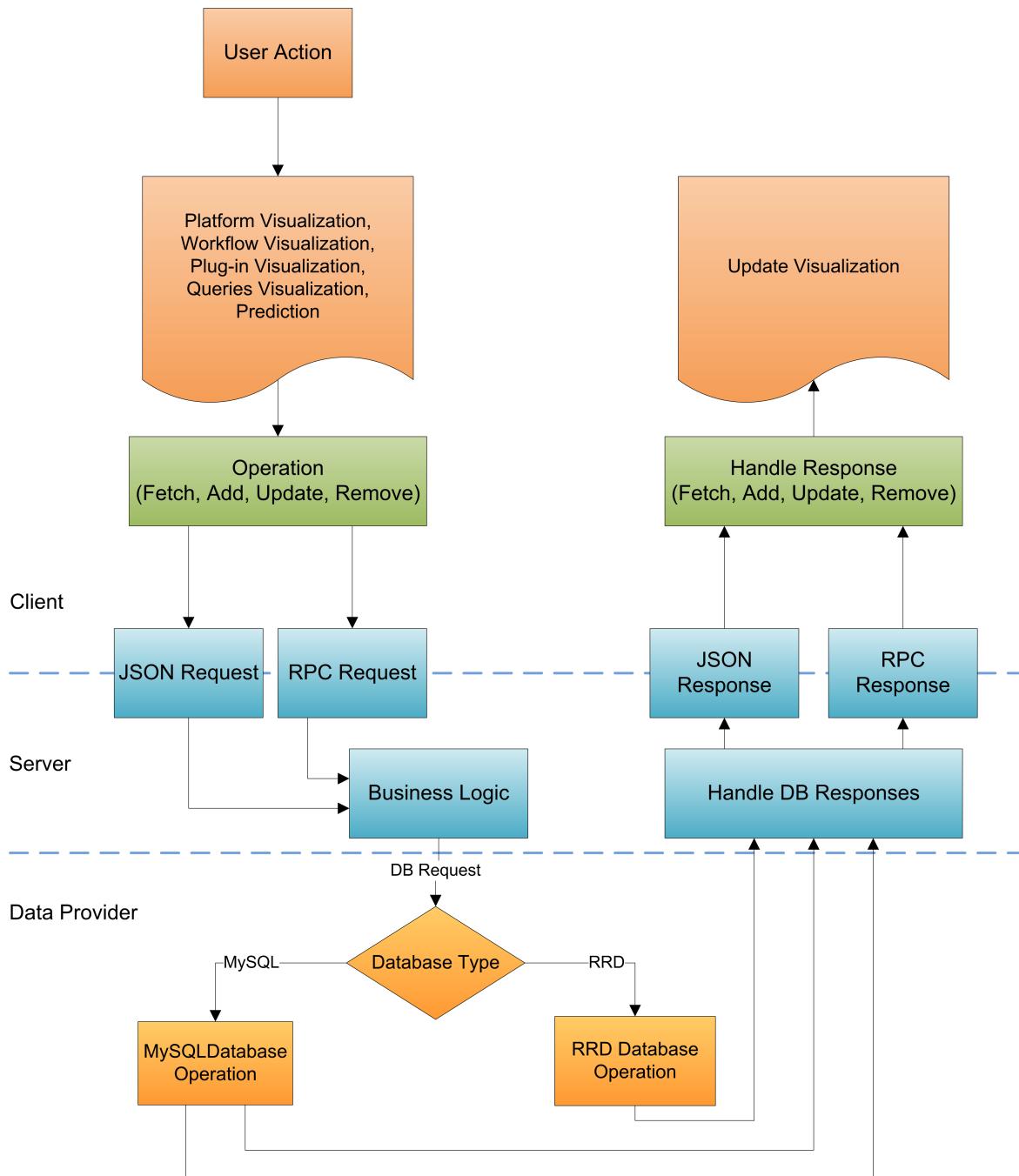


Figure 2.14: Visualization Flowchart

2.5.2 Graph based visualization of monitoring ontology and data

In the rest of this section we introduce a web based graphical tool that can be used to visualize graph structures such as ontologies. We use this tool to visualize the monitoring ontology that includes the metrics defined in [2] and also the instance data of the metrics. The tool is designed as a client server application, the client is used for visualization and the server supplies the data for the client. The client is a javascript application running into any browser supporting HTML5. The ontology is visualized into the client as a graph. The nodes of the graph are classes, object properties and data properties. The D3 library specialized in manipulating documents based on data is used to create interactive SVG graph and charts, see <http://mbostock.github.com/d3/>. Visualization of the metrics ontology using the tool is illustrated in Figure 2.15.

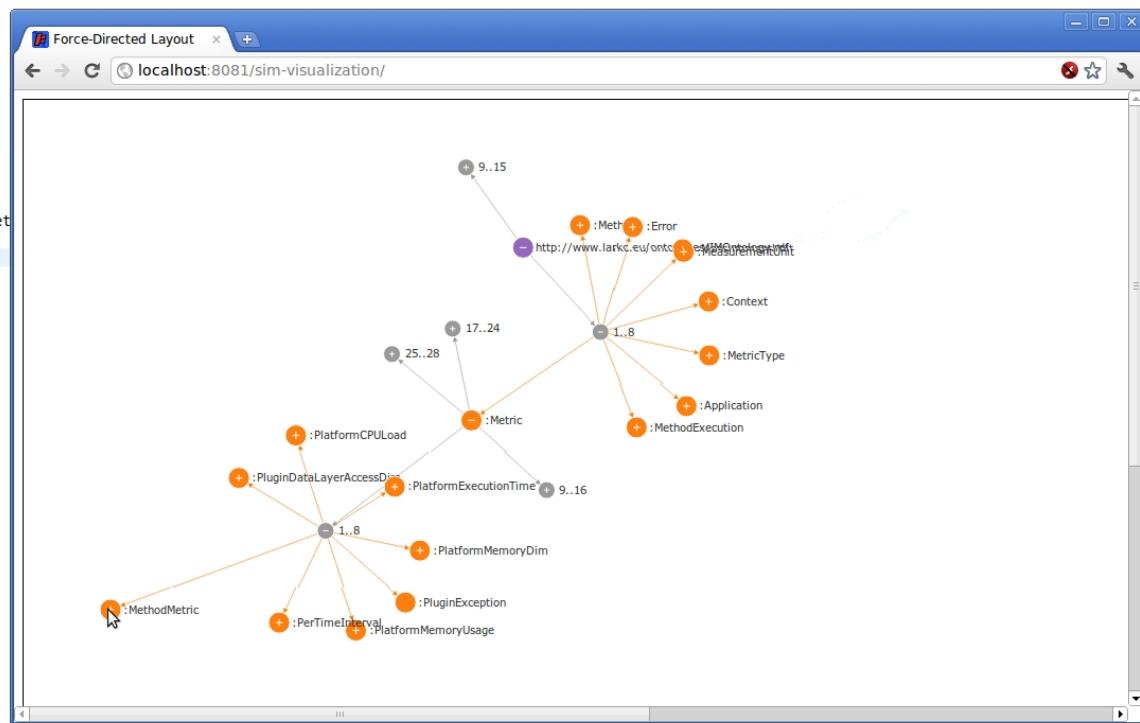


Figure 2.15: Graph based visualization of monitoring ontology and data - Visualization of metrics ontology

The navigation starts from one root element, which is the ontology. The children of the root element are all the classes that are no subclasses of any other class (we ignore here the possible inferences which would make any class a subclass of owl:Thing). One class can have the following children: other classes that directly subclass it, properties having this class as range or domain. To allow a better navigation of this graph we group into clusters children more than a configurable value. For example, if a class has 18 children and the cluster maximum size is 10 then before displaying the children we display 2 clusters, one which give access to the first 10 elements of the class and another one giving access to the other 8 children.

The client obtains the data to display from the server. The server is a web server following the REST (REpresentational State Transfer) technology and is im-

plemented using Restlet framework¹¹. The data transfer between client and server and vice versa is respecting JSON format¹². Most of the client calls are done asynchronously to assure a smooth visualization. On his end the server connects to a RDF repository (OWLIM or Sesame) for obtaining the ontology schema and instances. The data is obtaining using SPARQL interrogations and the Java library for this job is RDF2Go¹³. The source code of the tool is available at <https://github.com/semantic-im/sim-visualization>.

Visualization of the monitoring data is illustrated in Figure 2.16, Figure 2.17, Figure 2.18 and Figure 2.19. Figure 2.16 shows the panel that allows the user to select the metrics she/he is interested in. Figure 2.17 shows graphics of the selected metrics. Figure 2.18 shows graphics of the selected metrics for selected intervals. Finally, Figure 2.19 shows how to select the method metric type and method for which one would like to visualize the monitoring data.

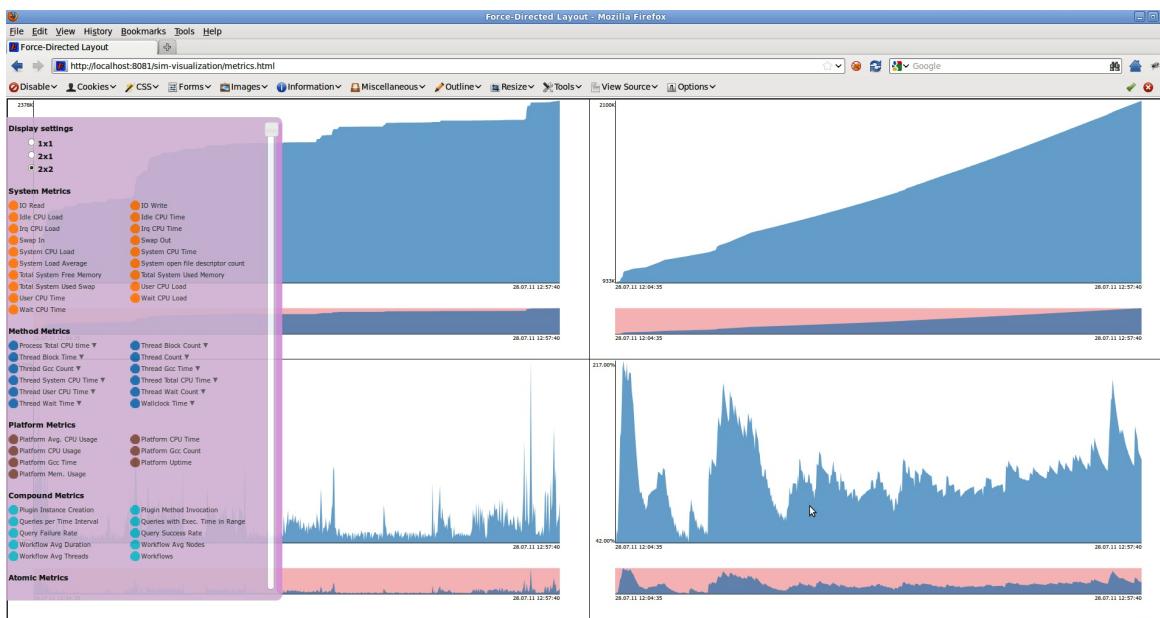


Figure 2.16: Graph based visualization of monitoring ontology and data - Panel for metrics selection and metrics charts

¹¹<http://www.restlet.org/>

¹²<http://www.json.org/>

¹³<http://semanticweb.org/wiki/RDF2Go>

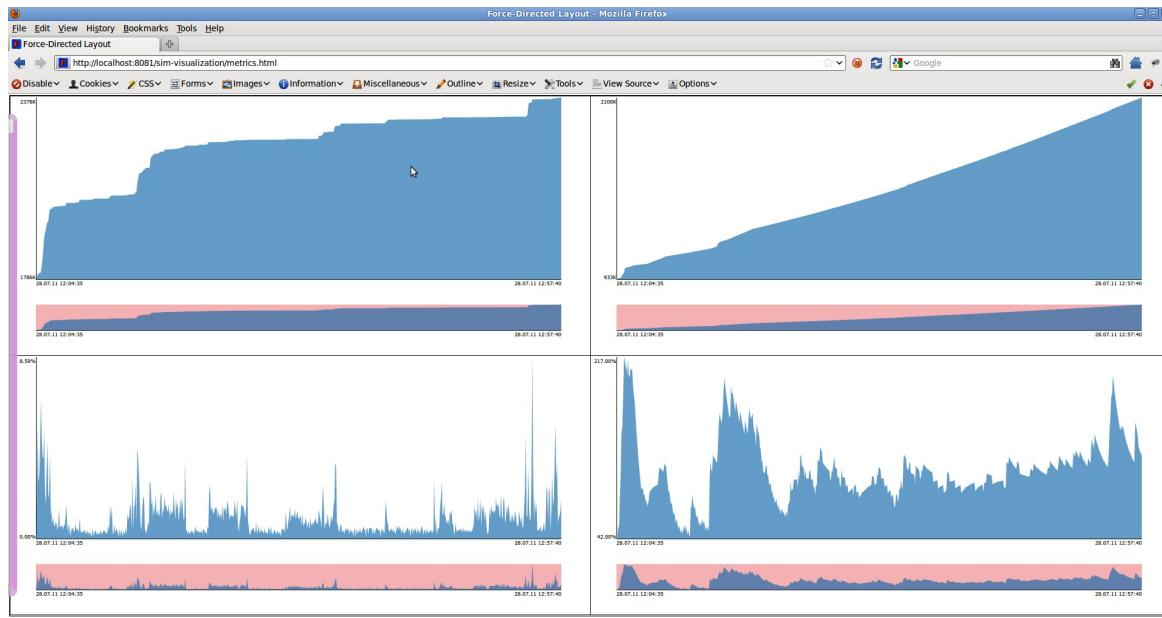


Figure 2.17: Graph based visualization of monitoring ontology and data - Metrics charts

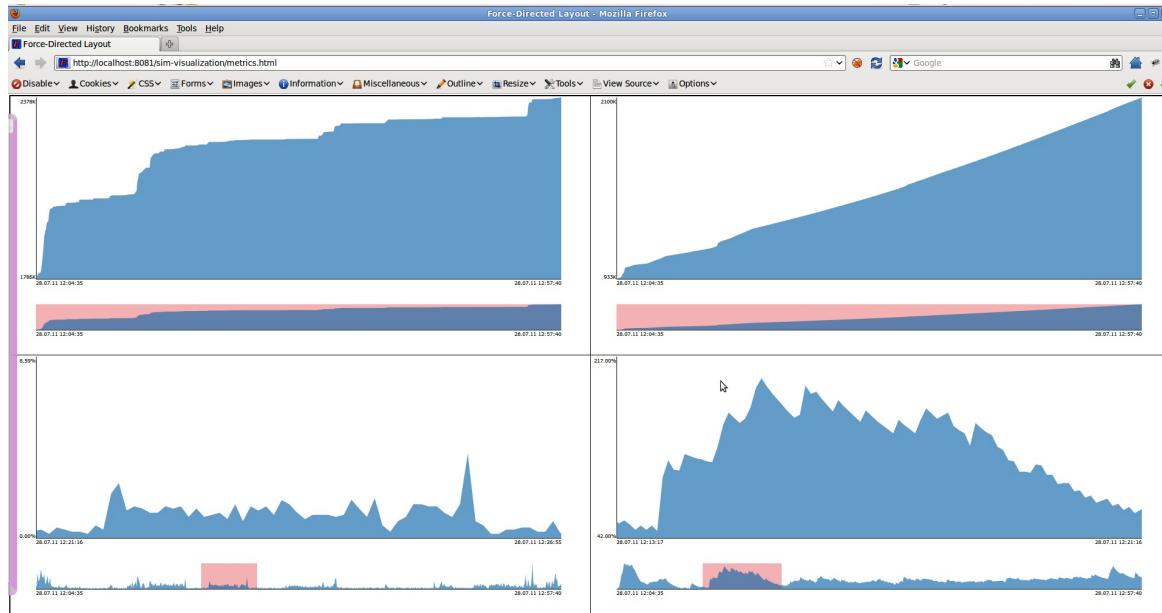


Figure 2.18: Graph based visualization of monitoring ontology and data - Metrics charts for selected intervals

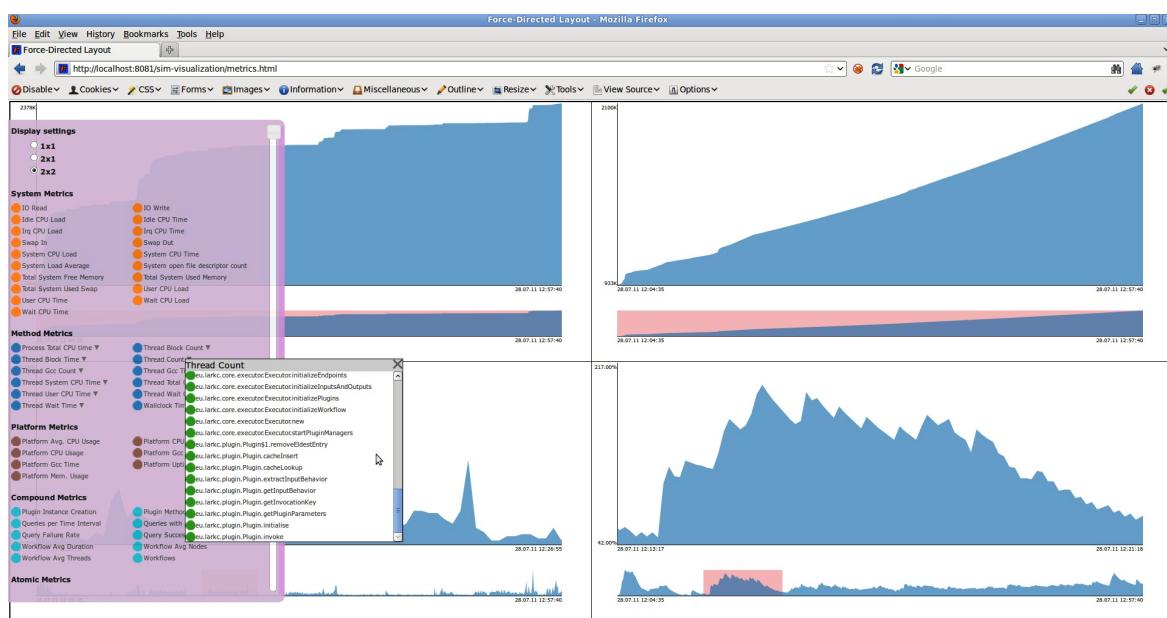


Figure 2.19: Graph based visualization of monitoring ontology and data - Selecting the method metric type and method

3. Installation and End-user guide

This section contains information on how the WP11 tools, namely instrumentation, relevance feedback and visualization tools, can be installed and used. This section is an updated of the initial guide provided in [1], Section 3.

3.1 Instrumentation

The LarKC platform depends on sim-instrumentation (a WP11 project hosted on <https://github.com/semantic-im/sim-instrumentation> that contains generic instrumentation and monitoring code) and *aspectjweaver javaagent* (part of AspectJ) which is used to enable LTW (Load Time Weaving). Specific code for instrumenting and monitoring the LarKC platform is available at <https://github.com/semantic-im/sim-instrumentation-larkc>. This code is made available also as part of the LarKC platform.

As mentioned in Section 2, instrumentation is done with the help of AspectJ¹, an aspect oriented extension to Java. Using AspectJ, SIM injects instrumentation code into specific locations, using byte-code manipulation. Customization of the AspectJ LTW can be done with the help of *META-INF/aop.xml* file which defines the aspects to apply on code at runtime and is read at startup by *aspectjweaver javaagent*.

3.1.1 How to enable SIM for LarKC

To enable instrumentation on LarKC, aspectjweaver.jar must be set as javaagent for the LarKC JVM. On command line (linux with maven) type:

```
java -Xmx512m -javaagent:$HOME/.m2/repository/org/aspectj/aspectjweaver/1.6.9/aspectjweaver-1.6.9.jar  
-jar larkc-platform.jar
```

where *\$HOME/.m2/repository* represents the maven repository, where aspectjweaver should be downloaded by maven when LarKC platform is build.

Another way of enabling instrumentation is with Eclipse for which you need to create a new Java Application Run Configuration following the steps:

1. Menu->Run->Run Configurations...
2. select Java Application
3. click New launch configuration button
4. for Project, click Browse and select the LarKC platform project
5. for main class enter: eu.larkc.core.Larkc
6. select Arguments tab and for VM arguments enter:

```
-Xmx512m -javaagent: ${system\ _property:user.home}  
/.m2/repository/org/aspectj/aspectjweaver/1.6.9/  
aspectjweaver-1.6.9.jar
```

where *\$system_property:user.home* returns the user home folder and *\$system_property:user.home/.m2/repository* represents the maven repository,

¹<http://www.eclipse.org/aspectj/downloads.php>

where aspectjweaver should be downloaded by maven when Larkc platform is build.

To view the measurements made by instrumentation sim-agent and sim-server applications when running you can download the binaries from <http://sim.softgress.com/>. Just download the binaries, extract and execute the run scripts. The results of measurements done by instrumentation and agent will be printed on the console by the agent and server applications. Agent system measurements are also exposed to JMX. You can view them by using jconsole (comes with java):

1. start jconsole
2. connect to sim-agent (sim-agent must be running)
3. open MBeans tab
4. select sim.agent->Agent->Attributes
5. click refresh to see updated values or double click any of the values to see a chart with the history of that value.

3.1.2 Understanding the results of instrumentation

Example of generic method execution measurement

```
2010-11-12 18:20:48,973 [Thread-3] INFO sim.agent.AgentHandler - 
MethodMetricsImpl [methodName=invoke,
className=eu.larkc.plugin.reason.SparqlQueryEvaluationReasoner,
context=, exception=null, endedWithError=false,
beginExecutionTime=1289578818723, endExecutionTime=1289578820508,
wallClockTime=1785, threadUserCpuTime=540, threadSystemCpuTime=40,
threadTotalCpuTime=580, threadCount=1, threadBlockCount=0,
threadBlockTime=0, threadWaitCount=0, threadWaitTime=0,
threadGccCount=1, threadGccTime=193, processTotalCpuTime=810]
```

Interpretation of method execution measurement

- What method was executed?
`eu.larkc.plugin.reason.SparqlQueryEvaluationReasoner.invoke`
- How long did this method take to execute (from the moment of calling this method to the moment of this method returning to the caller)?
`1785 ms (wallClockTime=1785)`
- Did this method call ended with an uncaught exception?
`No (endedWithError=false)`
- How many threads did this method create (directly or indirectly)?
`1 (threadCount=1)`
- How much user cpu time was spent by the thread executing this method?
`540 ms (threadUserCpuTime=540)`

- How much system cpu time (cpu time spent by OS to service this method) was spent by the thread executing this method?
40 ms (threadSystemCpuTime=40)
- Did this thread blocked (waiting for a monitor lock)? If yes, how many times and how much time did all take?
No. 0 times, 0 ms (threadBlockCount=0, threadBlockTime=0)
- Did this thread waited to be notified by other threads? If yes, how many times and how much time did all take?
No. 0 times, 0 ms (threadWaitCount=0, threadWaitTime=0)
- During execution of this method, was there a gcc? If yes, how many times and how much time did all take?
Yes. 1 times, 193 ms (threadGccCount=1, threadGccTime=193)
- How much cpu time was spent by the whole application during the execution of this method?
810 ms (processTotalCpuTime=810)
- Where does the difference between wallClockTime and processTotalCpuTime comes from?
processTotalCpuTime indicates cpu time effectively used by this application (the total time passed when this application was actually scheduled by the OS to execute). wallClockTime indicates the time elapsed between start of this method execution and end of this method execution (it does not differentiate if half of this time the application was not actually running - not scheduled by the OS to execute). So it would seem that 975 ms from the total 1785 ms this application was not running. This is explained by the fact that the machine where the test was run has only one CPU with one core and during the test other processes were scheduled by the OS to run.

Example of LarKC specific execution measurements

When using the workflows/foaf.rdf workflow and the default foaf “Frank van Harmelen” SPARQL query we get these measurements:

```
2011-01-27 04:05:05,179 [Thread-2] INFO sim.server.HttpCommunicationHandler
-
MethodMetricsImpl [creationTime=1296093873029, methodName=main,
className=eu.larkc.core.Larkc,
context=platformid=022660e2-ea8d-4df7-a0d6-410404ac414c,
tag=platform, name=PlatformStart,
exception=null, endedWithError=false,
beginExecutionTime=1296093873034, endExecutionTime=1296093896403,
wallClockTime=23369, threadUserCpuTime=220, threadSystemCpuTime=80,
threadTotalCpuTime=300, threadCount=4, threadBlockCount=0,
threadBlockTime=0, threadWaitCount=11, threadWaitTime=22899,
threadGccCount=17, threadGccTime=3545, processTotalCpuTime=18560]
2011-01-27 04:05:15,194 [Thread-2] INFO sim.server.HttpCommunicationHandler
-
MethodMetricsImpl [creationTime=1296093903709, methodName=<init>,
className=eu.larkc.core.executor.Executor,
```

```

context=platformid=022660e2-ea8d-4df7-a0d6-410404ac414c,
tag=workflow, name=WorkflowCreate,
workflowid=9ca86c9d-55bd-4e8e-9681-bf6ccb382ce7,
exception=null, endedWithError=false,
beginExecutionTime=1296093903709, endExecutionTime=1296093906923,
wallClockTime=3214, threadUserCpuTime=700, threadSystemCpuTime=40,
threadTotalCpuTime=740, threadCount=34, threadBlockCount=1,
threadBlockTime=166, threadWaitCount=13, threadWaitTime=935,
threadGccCount=1, threadGccTime=610, processTotalCpuTime=2540]
2011-01-27 04:05:20,207 [Thread-2] INFO sim.server.HttpCommunicationHandler
-
MethodMetricsImpl [creationTime=1296093913372, methodName=invoke,
className=eu.larkc.plugin.Plugin,
context=platformid=022660e2-ea8d-4df7-a0d6-410404ac414c,
queryid=4e9972e7-f09d-483e-ad8d-1a4f9bc6ae39,
tag=plugin, pluginName=urn:eu.larkc.plugin.hadoop.FileIdentifier,
name=PluginExecution,
pluginid=da45feee-54a7-4a77-81a1-d8571306f96f,
workflowid=9ca86c9d-55bd-4e8e-9681-bf6ccb382ce7,
exception=null, endedWithError=false,
beginExecutionTime=1296093913372, endExecutionTime=1296093913373,
wallClockTime=1, threadUserCpuTime=0, threadSystemCpuTime=0,
threadTotalCpuTime=0, threadCount=0, threadBlockCount=0,
threadBlockTime=0, threadWaitCount=0, threadWaitTime=0,
threadGccCount=0, threadGccTime=0, processTotalCpuTime=0]
2011-01-27 04:05:20,207 [Thread-2] INFO sim.server.HttpCommunicationHandler
-
MethodMetricsImpl [creationTime=1296093913536, methodName=invoke,
className=eu.larkc.plugin.Plugin,
context=platformid=022660e2-ea8d-4df7-a0d6-410404ac414c,
queryid=4e9972e7-f09d-483e-ad8d-1a4f9bc6ae39,
tag=plugin, pluginName=urn:eu.larkc.plugin.hadoop.HadoopRDFReader,
name=PluginExecution, pluginid=06ade4ea-97ea-4827-9104-0b410ddce05c,
workflowid=9ca86c9d-55bd-4e8e-9681-bf6ccb382ce7,
exception=null, endedWithError=false,
beginExecutionTime=1296093913536, endExecutionTime=1296093914816,
wallClockTime=1280, threadUserCpuTime=160, threadSystemCpuTime=0,
threadTotalCpuTime=160, threadCount=4, threadBlockCount=4,
threadBlockTime=0, threadWaitCount=4, threadWaitTime=141,
threadGccCount=1, threadGccTime=15, processTotalCpuTime=1110]
2011-01-27 04:05:20,208 [Thread-2] INFO sim.server.HttpCommunicationHandler
-
MethodMetricsImpl [creationTime=1296093914853, methodName=invoke,
className=eu.larkc.plugin.Plugin,
context=platformid=022660e2-ea8d-4df7-a0d6-410404ac414c,
queryid=4e9972e7-f09d-483e-ad8d-1a4f9bc6ae39,
tag=plugin, pluginName=urn:eu.larkc.plugin.reason.SparqlQueryEvaluationReasoner,
name=PluginExecution,
pluginid=18821527-6f47-4e8c-95d3-497fdf77ec12,
workflowid=9ca86c9d-55bd-4e8e-9681-bf6ccb382ce7,

```

```

exception=null, endedWithError=false,
beginExecutionTime=1296093914853, endExecutionTime=1296093915708,
wallClockTime=855, threadUserCpuTime=440, threadSystemCpuTime=20,
threadTotalCpuTime=460, threadCount=1, threadBlockCount=0,
threadBlockTime=0, threadWaitCount=0, threadWaitTime=0,
threadGccCount=1, threadGccTime=84, processTotalCpuTime=780]
2011-01-27 04:05:20,209 [Thread-2] INFO sim.server.HttpCommunicationHandler
-
MethodMetricsImpl [creationTime=1296093913106, methodName=handle,
className=eu.larkc.core.endpoint.sparql.SparqlHandler,
context=platformid=022660e2-ea8d-4df7-a0d6-410404ac414c,
queryid=4e9972e7-f09d-483e-ad8d-1a4f9bc6ae39, tag=query, name=QueryTotal,
workflowid=9ca86c9d-55bd-4e8e-9681-bf6ccb382ce7,
exception=null, endedWithError=false,
beginExecutionTime=1296093913106, endExecutionTime=1296093915920,
wallClockTime=2814, threadUserCpuTime=140, threadSystemCpuTime=20,
threadTotalCpuTime=160, threadCount=5, threadBlockCount=2,
threadBlockTime=1, threadWaitCount=5, threadWaitTime=2231,
threadGccCount=2, threadGccTime=99, processTotalCpuTime=2400]

```

All the above measurements mean that platform took 23.4 seconds to start, created 4 threads and generated 17 garbage collections that took 3.5 seconds (**red** markings in the listing). Workflow creation took 3.2 seconds and created 34 threads(**blue** markings in the listing). Total Query execution took 2.8 seconds and created 5 threads (**green** markings in the listing). Of this 1.3 seconds was spent by HadoopRDFReader plugin, which created 4 threads and 0.9 seconds was spent by SparqlQueryEvaluationReasoner which created 1 threads (**magenta** markings in the listing).

3.1.3 Extending SIM

Defining the classes which should be instrumented can be done in 3 ways:

1. add @sim.instrumentation.annotation.Instrument annotation to a class

```

@sim.instrumentation.annotation.Instrument
public abstract class Plugin implements MessageListener {
    ....
}

```

2. define a concrete-aspect for sim.instrumentation.aop.aspectj.AbstractMethodInterceptor into aop.xml

```

<concrete-aspect name="sim.instrumentation.aop.aspectj.larkc.InstrumentPlugin"
extends="sim.instrumentation.aop.aspectj.AbstractMethodInterceptor">
<pointcut name="methodExecution" expression="within(eu.larkc.plugin.Plugin)
AND execution(* *(..))"/>
</concrete-aspect>

```

3. using AspectJ to extend sim.instrumentation.aop.aspectj.AbstractMethodInterceptor (or define other custom aspects). A good example is sim.instrumentation.aop.aspectj.larkc.InstrumentPlugin which uses *beforeInvoke* and *afterInvoke* hooks to create an ExecutionFlowContext for the duration of the execution of the invoke method of the Plugin. In this way, all other instrumented methods that will be executed during the invoke method will be part of the created ExecutionFlowContext. Furthermore, using the JoinPoint from the *beforeInvoke* method, information about the name of the executing Plugin is extracted and placed into the created ExecutionFlowContext.

3.2 Visualization

The visualization module is a web-based, client server application. It can be installed and used on any computer.

The software programs that need to be installed in order to start using the visualization module are:

1. Apache Tomcat version 7.0.6 or later, or other compatible application server².
2. MySQL Server version 5.0.24³ or later.

The source code for visualization can be downloaded from: <https://github.com/semantic-im/sim-vis>. It contains the following:

- MySQLDb
- RestServer
- Visualization_Metrics
- Visualization_Prediction

The steps that need to be done are:

1. Install mysql db server
 - <http://dev.mysql.com/doc/refman/5.1/en/installing.html>
 - Create an empty data base with the name “larkc” (default user is “root” and password is “1111”)
 - Import sql file: larkc-core-2011-01-28.sql into your database
2. For RestServer:
 - (a) Run the rest server with the input arguments:
`java -cp .;..\lib\jrobin-1.5.9.1.jar;..\lib\org.restlet.jar;..\lib\rrd4j-2.0.5.jar \Rrd4jJSONServer "path\to\the\RestServer\!rrds"`
 - (b) Add the following dependencies to the project: the .jar files in RestServer \lib.

²<http://tomcat.apache.org/>

³<http://dev.mysql.com/>

3. Visualization_Metrics and Visualization_Prediction

- (a) For Visualization_metrics it is recommended to run Eclipse with GWT plugin installed.
- (b) You should add the external libraries the following dependencies: `lib\ofcgwt.jar` and `lib\smartgwt.jar`.
- (c) For now visualization metrics and the prediction page are two separate projects.

4. Running all the projects:

- (a) In order to run the project the mysql server should be running.
- (b) Start RestServer
- (c) Run Visualization components (Metrics and Prediction) - recommended to use Eclipse to run them because it is easier to debug them.

3.3 Relevance Feedback

3.3.1 Compiling the application

This module has two main components - off-line training and on-line prediction. The off-line training can be downloaded from <https://github.com/semantic-im/sim-rf/RFJava>.

Requirements:

1. Netbeans IDE version 7.0 (<http://netbeans.org/index.html>)
2. Java jdk1.6.0_22.

To run the off-line training one should follow the steps described below:

1. Download the source code from <https://github.com/semantic-im/sim-rf>.
2. Start Netbeans and choose “Open Project”. Choose the directory where you have downloaded the source code - it should be RFJava.
3. The special dependencies that may be needed are found in the lib directory of the project (but no special settings are needed. The code should compile without adding the dependencies).
4. Run the project from the Netbeans IDE.
5. The screen depicted in Figure 3.1 should appear. It has three tabs – “Load” (loads a data set for training), “Machine Learning” (applies a machine learning algorithm on the loaded data) and “Test” (applies the trained machine learning model on a test query).

3.3.2 Running an example

The first step that needs to be done for the off-line training module is to load some data that will be used for training. We have generated some sample data from the workflows that have been run. The sets are included in the “arff” directory. In order to load the data, one needs to select the “Load” tab and from there press the button “Choose data source”. A popup dialog will be displayed. It gives the possibility to choose the arff file.

Two files are available “metricsMethod.arff” (used by “Clustering and regression” or by “Kernel regression” methods from the “Machine learning” tab) and “query_allWF.arff” (that can be used by “Best configuration” method from the “Machine learning” tab).

After the file “metricsMethod.arff” is loaded, the application looks like in Figure 3.1.

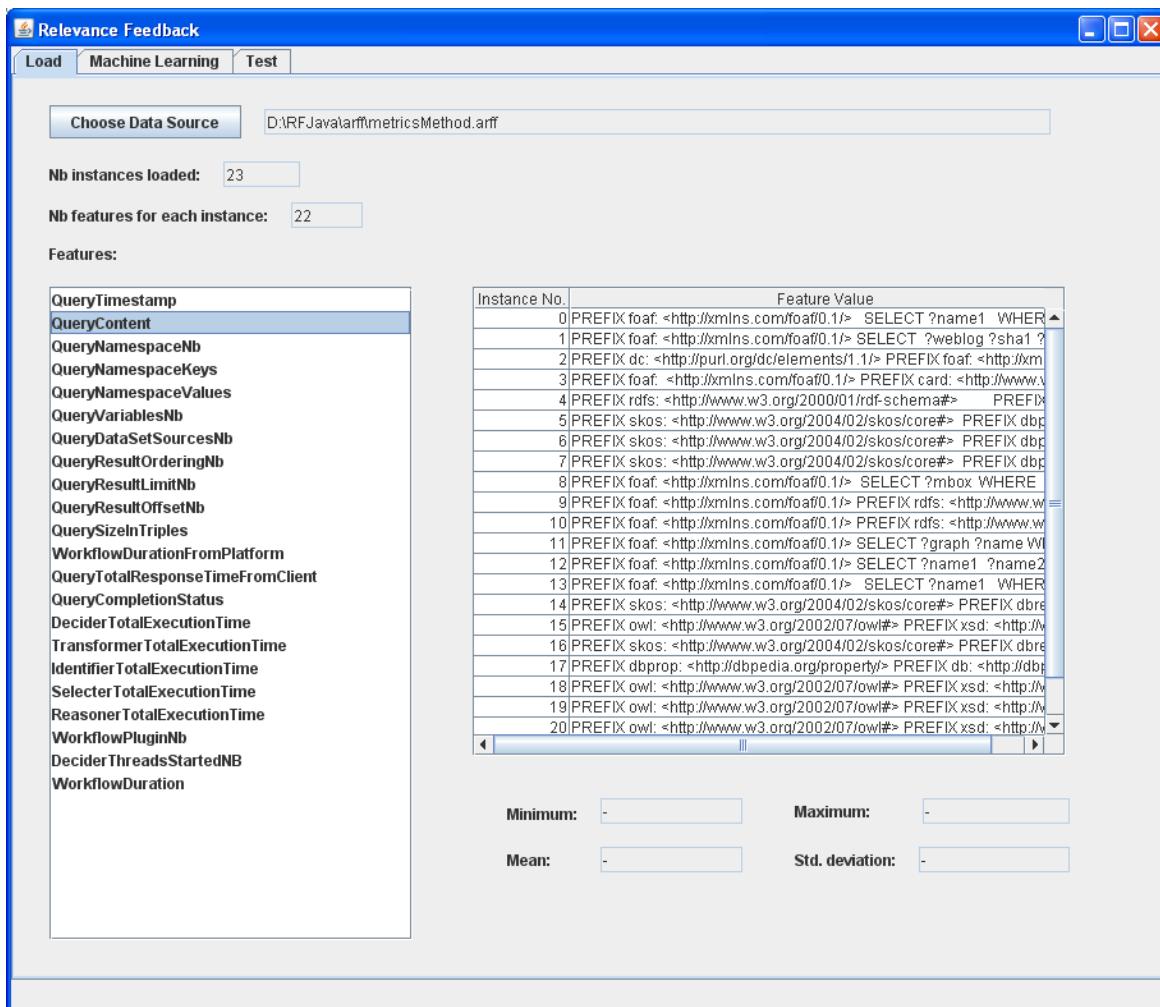


Figure 3.1: Data loading in the off-line RF training application

The second step consists in choosing the machine learning method that is applied on the loaded data. For this, select the “Machine Learning” tab. For example if the file “metricsMethod.arff” has been loaded, then one should check the “Clustering and Regression” item. Then press the button “Build prediction model”. When the training is finished a message will be displayed in the “Status” text area. If the training is successful the model can be saved by pressing the button “Save model” and a popup dialog will allow the saving of the model on the disk. Figure 3.2 shows the described

process. When performing steps one and two mentioned previously, an important

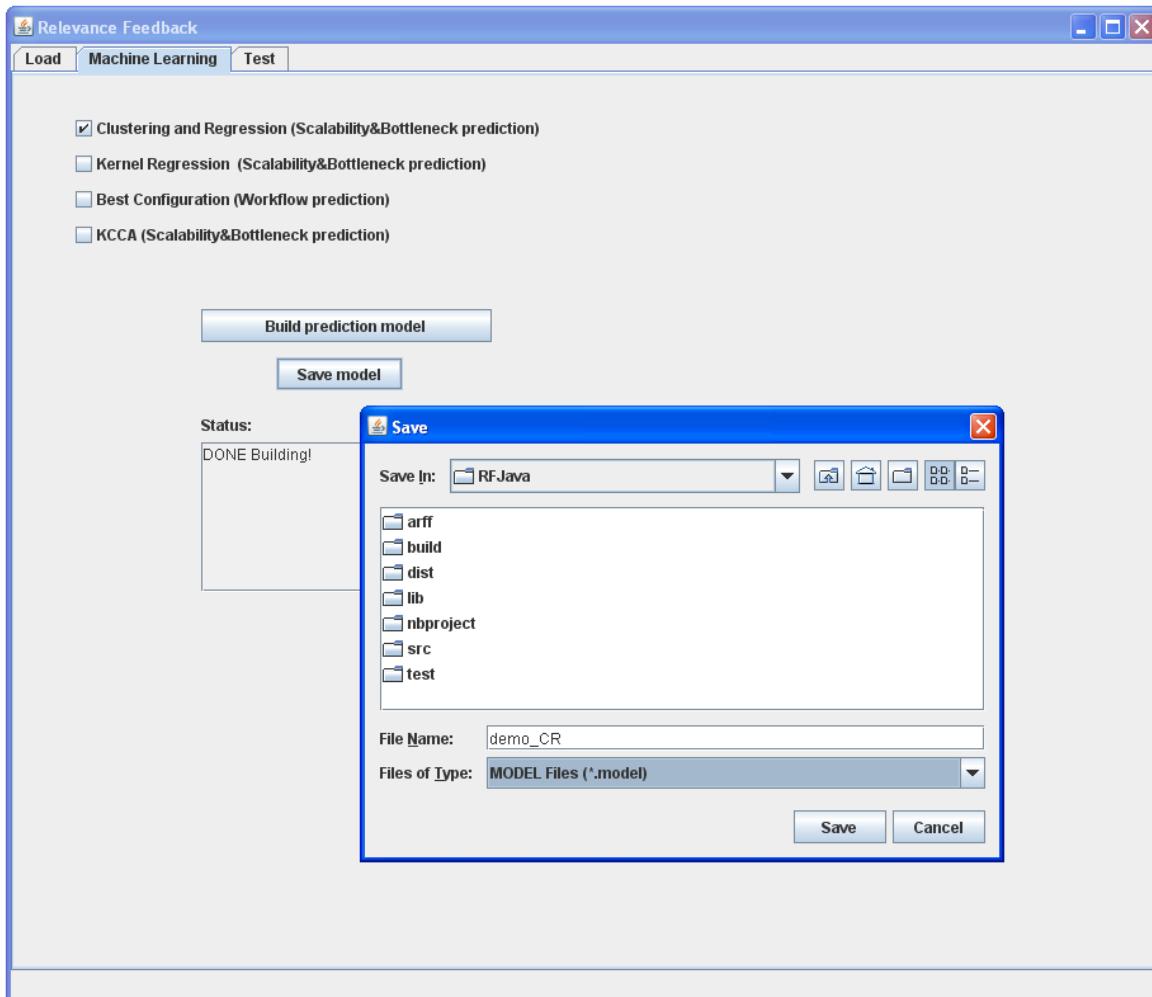


Figure 3.2: Building and saving a RF machine learning model

remark is that the model can be applied automatically (it does not need to be loaded). For example when choosing the “Test” tab one is able to test the model on new input data that is in the form of a SPARQL query (the text area “Query” from the interface). By pressing the “Predict” button the trained model is applied on the new query and the results are displayed in the text area below. Figure 3.3 shows the prediction results.

Steps one and two can be skipped if one wants to directly test a previously stored model. For example, consider we have previously saved a “Best configuration” model. This model can be loaded in the “Test” tab by pressing “Load prediction model” button and choosing the file saved on the disk. If the model is loaded successfully a message is displayed in the “Status” text box. Next the model can be applied on a test query by pressing the “Predict” button. The results are displayed in the text area below. Figure 3.4 shows the prediction results.

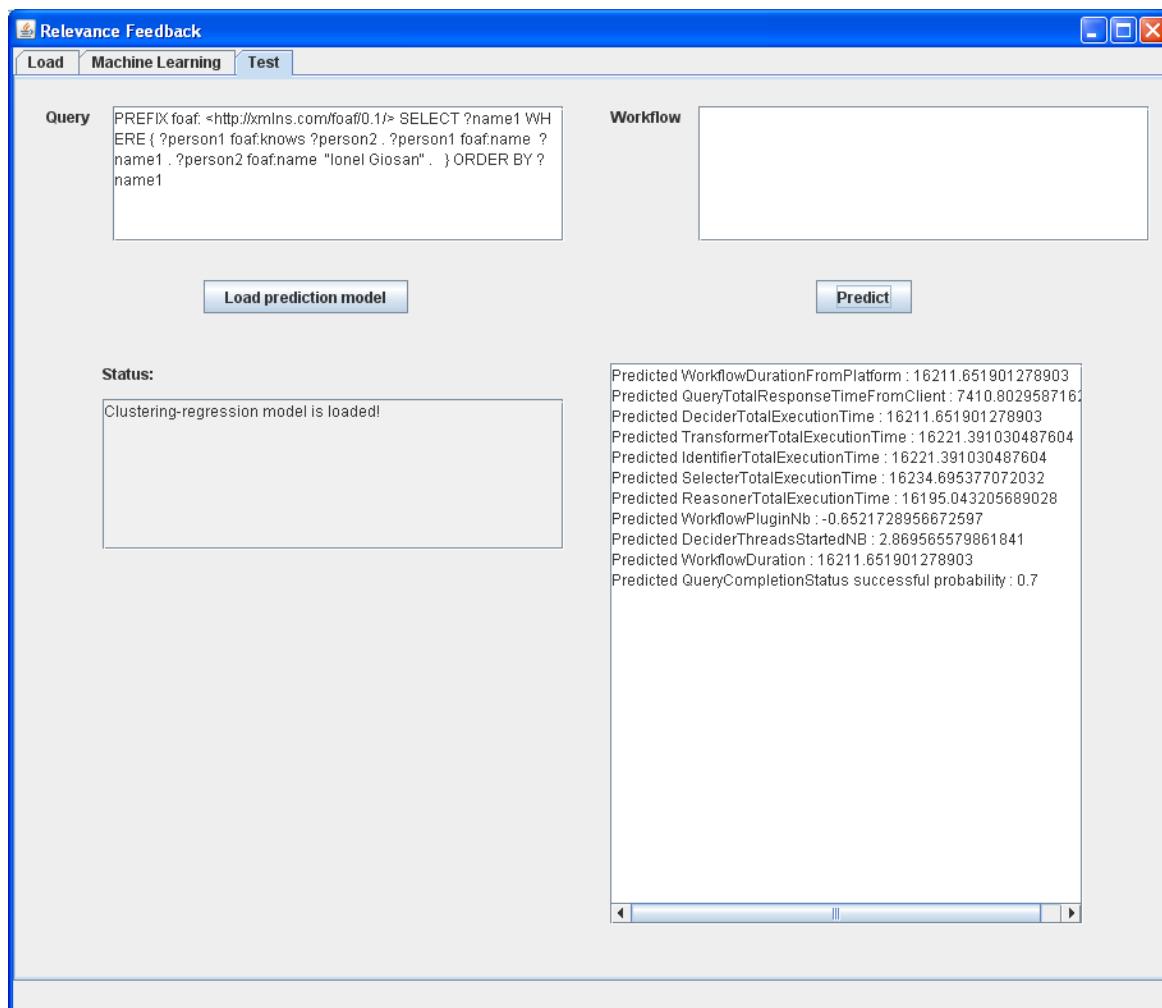


Figure 3.3: Using the RF model for predicting the parameters of a query.

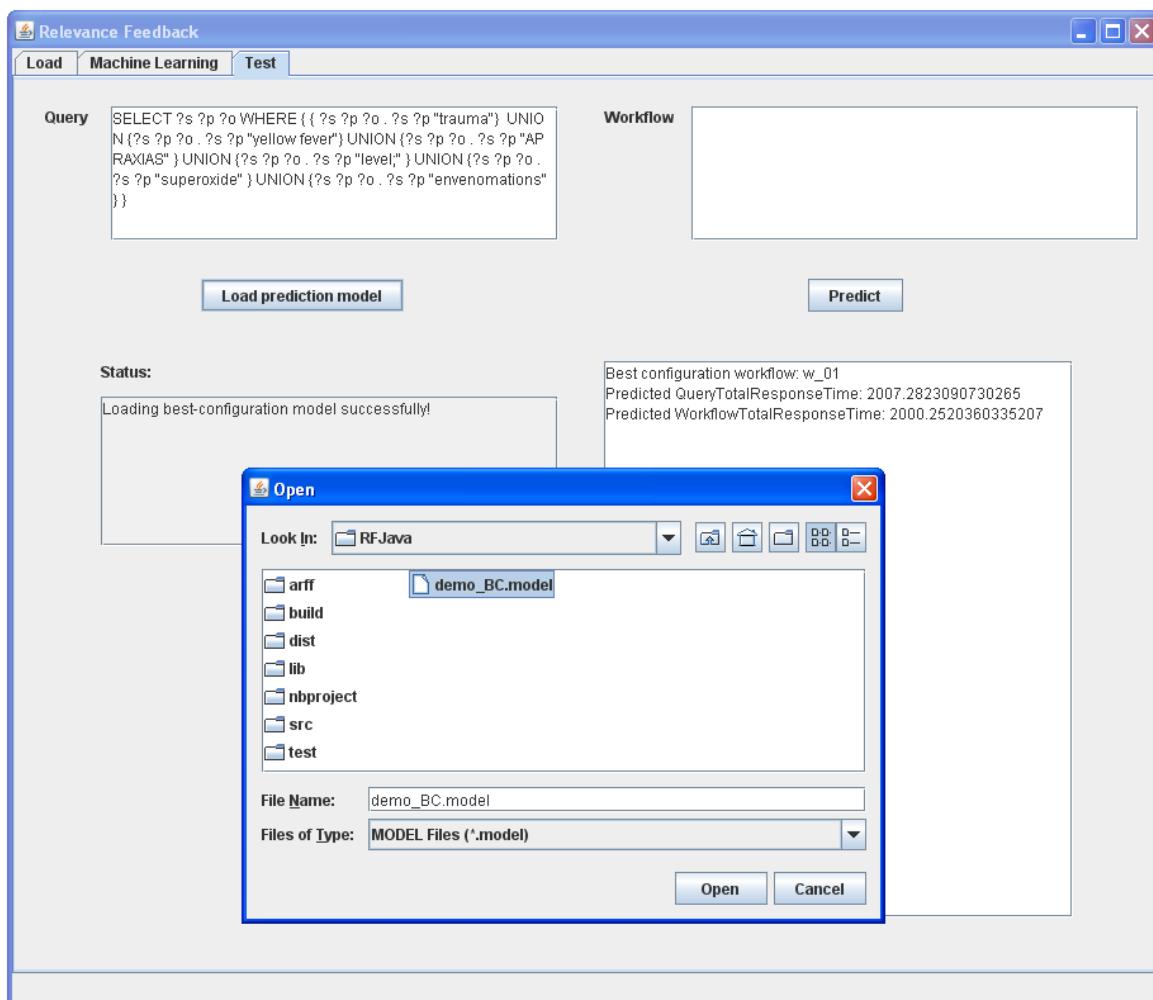


Figure 3.4: Loading and testing a Best configuration RF model

4. Using SIM tools for instrumenting and monitoring LarkC plugins and workflows

4.1 Instrumented Workflows

The experiments we have done so far are based on the monitoring and instrumentation of the following workflows, including the constituents plugins and the queries used by these workflows:

- LLD_REASONIG - consisted of the composition of two plug-ins: LLDReasoner, SOStoVBtransformer
- QUERY_EXPANSION – consists in the composition of the following plug-ins: RISearchPlugin , QueryExpansion, LLDReasoner, SOStoVBTransformer.
- CRIONReasoner – comprises the plug-ins: CRIONReasonerPlugin, SOStoVB-transformer
- FactForge – includes the plug-ins: RDFReader, SparqlQueryEvaluationReasoner, NewFileIdentifier, SPARQLToTriplepatternTransformer, SindiceIdentifier
- KeywordReasoner2 – KeywordReasonerPlugin2, SOStoVBtransformer
- KeywordReasoner3 – KeywordReasonerPlugin3, SOStoVBtransformer
- OWLAPIReasoner – OWLAPIReasonerPlugin
- PIONReasoner – PIONReasonerPlugin, SOStoVBtransformer
- Simple – RDFReader, SparqlQueryEvaluationReasoner, NewFileIdentifier
- Sindice – RDFReader, SparqlQueryEvaluationReasoner, NewFileIdentifier, SPARQLToTriplepatternTransformer, SindiceIdentifier
- SPARQLDLReasoner – SPARQLDLReasonerPlugin
- TripIt – TripItPlugin

In order to capture several aspects for visualization and monitoring we have artificially varied the plug-ins that formed the workflows above and the Executor. The variations consist in adding additional delay (by Thread.sleep) and adding code to increase the CPU load.

We have generated configurations of the same workflows with these modifications in the plug-ins or in the Executor. Taking the LLD_REASONIG workflow as an example, the following configurations were obtained:

- w_01 – the workflow was run as it is, with no modifications;
- w_02 – the workflow was run with an increase in execution time for Executor (path is platform\src\main\java\eu\larkc\core\executor\Executor.java)(no modifications at plug-in level); The method that we have modified was: execute(SetOfStatements query, String pathId). The platform should be recompiled after the modification.

- w_03 – the workflow was run with an increase in CPU load for Executor (no modifications at plug-in level); We have modified the method execute(SetOfStatements query, String pathId) from Executor.java. The platform should be recompiled after the modification.
- w_04 – the workflow was run with an additional execution time added to the LLDReasoner plug-in (plugins\LLDReasoner\src\main\java\eu\larkc\plugin); In method ‘invokeInternal(SetOfStatements input)’ we have added the code:

```
try {
//Pause for 1 seconds
Thread.sleep(1000);
} catch (InterruptedException ex) {
ex.printStackTrace();
}
```

The plug-in should be recompiled after the modification.

- w_05 – the workflow was run with an increase in CPU load for the LLDReasoner plug-in; In the method ‘invokeInternal(SetOfStatements input)’ we have added the code:

```
final int NUM_TESTS = 1000;
int milliseconds = 500;
for (int i = 0; i < NUM_TESTS; i++) {
long sleepTime = milliseconds*100000L; // convert to nanoseconds
long startTime = System.nanoTime();
while ((System.nanoTime() - startTime) < sleepTime) {}
```

The plug-in should be recompiled after the modification.

- w_06 – the workflow was run with an increase in CPU load for both plug-ins (method invoke from eu.core.larkc.plugins.Plugin); The platform should be recompiled after the modification.

4.2 Visualization for instrumented workflows

The following metrics are used by the visualization component to graphically display to the user the monitoring information about instrumented queries executions, plugins, workflows and platform: QuerySizeInCharacters, QueryNamespaceNb, QueryVariablesNb, QueryDataSetSourcesNb, QueryOperatorsNb, QueryResultOrderingNb, QueryResultLimitNb, QueryResultOffsetNb, QueryResultSizeInCharacters, QueryTotalResponseTime, QueryProcessTotalCPUTime, QueryThreadTotalCPUTime, QueryThreadUserCPUTime, QueryThreadSystemCPUTime, QueryThreadCount, QueryThreadBlockCount, QueryThreadBlockTime, QueryThreadWaitCount, QueryThreadWaitTime, QueryThreadGccCount, QueryThreadGccTime, QueryContextInstanceFromWorkflow, WorkflowNumberOfWorkPlugins, WorkflowTotalResponseTime, WorkflowProcessTotalCPUTime, WorkflowThreadTotalCPUTime, WorkflowThreadUserCPUTime, WorkflowThreadSystemCPUTime, WorkflowThreadCount, WorkflowThreadBlockCount,

WorkflowThreadBlockTime, WorkflowThreadWaitCount, WorkflowThreadWaitTime, WorkflowThreadGccCount, WorkflowThreadGccTime and all the generic system and method metrics (e.g WaitCPUTime, UserCPUload,etc.).

The visualization results are based on information provided by the instrumentation and monitoring module. Figure 4.1 presents the visualization of the plugins (LLDReasoner, SOStoVBtransformer) corresponding to a selected workflow instance and a set of queries that were executed. It can be observed that for a single workflow there are many uniquely identified instances.

Id	Name
1c40b93b-6375-4a56-b6a4-432378b89b5e	eu.larkc.plugin.LLDReasoner;eu.larkc.plugin.SOStoVBtransformer;
2aaad516-a8ef-4511-8ffb-3e5d076f3d1a	eu.larkc.plugin.LLDReasoner;eu.larkc.plugin.SOStoVBtransformer;
d93d8a2c-db56-4f79-8eeef-5f59be56d9de	eu.larkc.plugin.LLDReasoner;eu.larkc.plugin.SOStoVBtransformer;
Plugin Name	
eu.larkc.plugin.LLDReasoner	
eu.larkc.plugin.SOStoVBtransformer	
Id Query	Query Context
00041fcf-8324-4549-b024-8fd4721a616b	SELECT ?s ?p ?o WHERE { { ?s ?p ?o . ?s ?p "zoophilia"} }
0017ddfc-f038-4e35-82c2-276f9df5663f	SELECT ?s ?p ?o WHERE { { ?s ?p ?o . ?s ?p "insomnia"} }
019471ac-714d-41ed-b8e0-82aadb7ebddd	SELECT ?s ?p ?o WHERE { { ?s ?p ?o . ?s ?p "Bartonella"} UNION {?s ?p ?o . ?s ?p "arthritis"} UNION {?s ?p ?o . ?s ?p "APRAXIAS"} UNION {?s ?p ?o . ?s ?p "level;" } UNION {?s ?p ?o . ?s ?p "superoxide" } }
01f00b9d-cea2-4a3e-8218-3cd868f9f464	SELECT ?s ?p ?o WHERE { { ?s ?p ?o . ?s ?p "harclepsy"} }
0297d006a-7cdb-4c12-a2bc-9fa29758921d	SELECT ?s ?p ?o WHERE { { ?s ?p ?o . ?s ?p "trauma"} UNION {?s ?p ?o . ?s ?p "Acrophobia"} UNION {?s ?p ?o . ?s ?p "APRAXIAS"} UNION {?s ?p ?o . ?s ?p "level;" } UNION {?s ?p ?o . ?s ?p "superoxide" } UNION {?s ?p ?o . ?s ?p "envenomations" } }

Figure 4.1: Plugins and queries corresponding to the LLD _ REASONING workflow

Figure 4.2 illustrates the obtained metrics for a selected query.

Figure 4.3 introduces a new version of the Visualization component that uses Liferay portal platform as the application aggregator and content management system. The metrics that are stored as time-series values are represented as a separate portlet that is integrated into the portal environment according to JSR-168/JSR-286 specification. As future development we are intended to develop new visualization functionalities and to adapt the existing parts to the portlet specification.

4.3 Relevance Feedback for instrumented workflows

Currently, the relevance feedback component considers the following metrics: QuerySizeInCharacters, QueryNamespaceNb, QueryVariablesNb, QueryDataSetSourcesNb, QueryOperatorsNb, QueryResultOrderingNb, QueryResultLimitNb, QueryResultOffsetNb, QueryResultSizeInCharacters, QueryTotalResponseTime, QueryProcessTotalCPUTime, QueryThreadTotalCPUTime, QueryThreadUserCPUTime, QueryThreadSystemCPUTime, QueryThreadCount, QueryThreadBlockCount, QueryThreadBlockTime, QueryThreadWaitCount, QueryThreadWaitTime, QueryThreadGccCount, QueryThreadGccTime, QueryContextInstanceFromWorkflow, WorkflowNumberOfWorkflows, Work-

Id Query		Query Context
00041fcf-8324-4549-b024-6fd4721a616b		SELECT ?s ?p ?o WHERE { { ?s ?p ?o . ?s ?p "zoophilia"} }
0017ddfc-f038-4e35-82c2-276f9df5663f		SELECT ?s ?p ?o WHERE { { ?s ?p ?o . ?s ?p "Insomnia"} }
019471ac-714d-41ed-b8e0-82aadb7ebddd		SELECT ?s ?p ?o WHERE { { ?s ?p ?o . ?s ?p "Bartonella"} UNION {?s ?p ?o . ?s ?p "arthritis"} UNION {?s ?p ?o . ?s ?p "APRAXIAS"} UNION {?s ?p ?o . ?s ?p "level;" } UNION {?s ?p ?o . ?s ?p "superoxide" } }
01f00b9d-cea2-4a3e-8218-3cd868f9f464		SELECT ?s ?p ?o WHERE { { ?s ?p ?o . ?s ?p "harclepsy"} }
0297006a-7fdb-4c12-a2bc-9fa29758921d		SELECT ?s ?p ?o WHERE { { ?s ?p ?o . ?s ?p "trauma"} UNION {?s ?p ?o . ?s ?p "Acrophobia"} UNION {?s ?p ?o . ?s ?p "AFRAXIAS"} UNION {?s ?p ?o . ?s ?p "level;" } UNION {?s ?p ?o . ?s ?p "superoxide" } UNION {?s ?p ?o . ?s ?p "envenomations" } }
Timestamp	Metric Name	Metric Value
2011-07-05 16:20:31.0	Query Content	SELECT ?s ?p ?o WHERE { { ?s ?p ?o . ?s ?p 'Insomnia' } }
2011-07-05 16:20:31.0	Query ErrorStatus	false
2011-07-05 16:20:31.0	Query SizeInCharacters	56
2011-07-05 16:20:31.0	Query Result SizeInCharacters	32567
2011-07-05 16:20:31.0	Query VariablesNb	3
2011-07-05 16:20:31.0	Query TotalResponseTime	2000
2011-07-05 16:20:31.0	Query ThreadWaitCount	2
2011-07-05 16:20:31.0	Query ThreadWaitTime	100.0

Figure 4.2: Query metrics

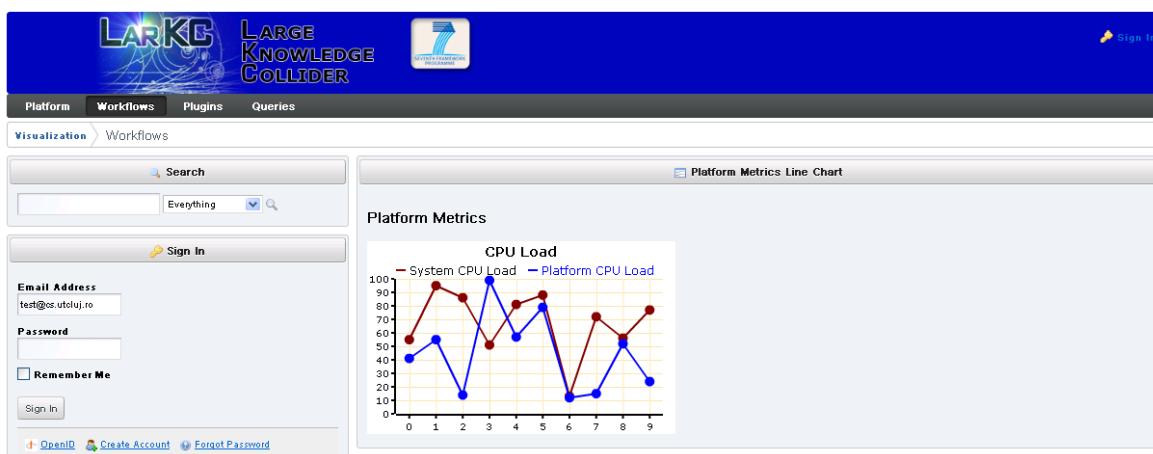


Figure 4.3: Metrics that are stored as time-series values. Portlet representation.

flowTotalResponseTime, WorkflowProcessTotalCPUTime, WorkflowThreadTotalCPUTime, WorkflowThreadUserCPUTime, WorkflowThreadSystemCPUTime, WorkflowThreadCount, WorkflowThreadBlockCount, WorkflowThreadBlockTime, WorkflowThreadWaitCount, WorkflowThreadWaitTime, WorkflowThreadGccCount, WorkflowThreadGccTime.

We will exemplify the work on an arff file that contains three workflow configurations (w_01, w_02, w_03). For these workflows we run two types of queries:

1. Example of type 1 queries:

```
SELECT ?s ?p ?o WHERE {
  {?s ?p ?o . ?s ?p "myopia"}
}
```

2. Example of type 2 queries:

```
SELECT ?s ?p ?o WHERE {
  { ?s ?p ?o . ?s ?p "trauma"}
  UNION {?s ?p ?o . ?s ?p "diabetes"}
  UNION {?s ?p ?o . ?s ?p "APRAXIAS" }
  UNION {?s ?p ?o . ?s ?p "level;" }
  UNION {?s ?p ?o . ?s ?p "superoxide" }
  UNION {?s ?p ?o . ?s ?p "envenomations" } }
```

For w_01 we run 248 queries, for w_02 we have run 250 queries and for w_03 we run 192 queries. Most of the queries were similar for the three workflow configurations.

On this data we wanted to find the best workflow configuration for a given input query and for it predict some output metric values.

Given the instrumentation data generated when running the queries for the three workflow configurations we have applied a clustering step. Two clusters were formed, the first contained 393 instances and the second contained 297 instances. We noticed that the two clusters correspond to the two types of queries we have given. The clustering was done based on the metrics collected for the SPARQL queries.

Inside each cluster we performed another grouping operation based on the workflow configuration parameters. This resulted in:

- for cluster 1: 93 instances for w_01, 151 instances for w_02 and 149 instances for w_03.
- for cluster 2: 99 instances for w_01, 99 instances for w_02 and 99 instances for w_03.

Also, for each cluster we have build a prediction model (using linear regression and kernel regression). Next, we have applied our model on several test input queries. For example, for the test query:

```
SELECT ?s ?p ?o
WHERE { { ?s ?p ?o . ?s ?p "myopia"}
UNION {?s ?p ?o . ?s ?p "yellow fever"}
UNION {?s ?p ?o . ?s ?p "APRAXIAS" }
UNION {?s ?p ?o . ?s ?p "level;" }
UNION {?s ?p ?o . ?s ?p "superoxide" }
UNION {?s ?p ?o . ?s ?p "envenomations" } }
```

The suggested best configuration was with w_01 (which is correct, as w_01 runs in a minimum amount of time and has a low CPU load). Also, for it the predicted total execution time was of 2000.2553795891301 ms - which was close to the actual execution time. More examples and test queries can be experimented using the demo version.

5. Conclusion

This deliverable reports on the development, since the first release, of the instrumentation and monitoring solution for LarKC, which we call *Semantic Instrumentation and Monitoring (SIM)*. We introduced and detailed the updates of each component, namely *instrumentation mechanism*, *profiling agents*, *server*, *visualization* and *relevance feedback*, in terms of architecture and implementation. The installation and user guide provided in [1] was updated to reflect the latest developments, providing details on how to install and use SIM components, and also how to instrument LarKC plug-ins and workflows. SIM is also well integrated in the overall LarKC architecture and platform, updates on LarKC instrumentation and monitoring being released regularly with the LarKC platform. The current version of SIM instrumentation and monitoring, via the instrumentation mechanism supports a large set of metrics, including methods, system, atomic and compound metrics. The profiling agents support the collection and buffering of data received from the instrumentation mechanism. The server component stores the monitoring data in RDF and implements a wide range of compound metrics generations methods. The instrumentation and monitoring solution also includes advanced visualization and relevance feedback functionality uses the metrics collected by instrumentation, agents and stored on the server. We also reported on the set of workflows and their plugins that are instrumented and how visualization and relevance feedback components are using the monitoring data obtained by running these workflows and plugins.

REFERENCES

- [1] I. Toma, R. Brehar, S. Nedevschi, M. Chezan, S. Bota, I. Giosan, M. Negru, and A. Vatavu, “Instrumentation and monitoring platform - design, architecture specification and first prototype,” LarKC Project Deliverable, Tech. Rep. D11.2, 2011.
- [2] I. Toma, R. Brehar, S. Bota, M. Negru, A. Vatavu, and M. Chezan, “Larkc metrics ontology,” LarKC Project Deliverable, Tech. Rep. D11.1.2, 2010.
- [3] R. Brehar, I. Giosan, M. Negru, A. Vatavu, I. Toma, and C. Vicas, “Visualization tool and relevance feedback for instrumentation and monitoring,” LarKC Project Deliverable, Tech. Rep. D11.3, 2011.