



LarKC

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

FP7 – 215535

D5.3.1 Requirements Analysis and report on lessons learned during prototyping

**Coordinator: Michael Witbrock, Blaž Fortuna, Luka Bradeško,
CycEur; Georgina Gallizo, HLRS.**

**With contributions from: Mick Kerrigan, Barry Bishop, UIBK;
Frank van Harmelen, Annete ten Teije, Eyal Oren, VUA; Vassil
Momtchev, Ontotext; Axel Tenschert, Alexey Cheptsov, Sabine
Roller, HLRS**

**Quality Assessor: Volker Tresp, SIEMENS
Quality Controller: Georgina Gallizo, HLRS**

Document Identifier:	LarKC/2008/D5.3.1 /v1.0
Class Deliverable:	LarKC EU-IST-2008-215535
Version:	0.11
Date:	28.05.2009
State:	Draft
Distribution:	Public



EXECUTIVE SUMMARY

The main goal of this deliverable is to identify the requirements of the LarKC Platform in terms of the features and capabilities that it must offer, its intended behaviour and interoperability with other LarKC components (specifically the LarKC Plug-ins, to be realised in WP2, 3 and 4) and its intended usage scenarios.

After the first fourteen (14) months of the project a wide set of requirements has been identified and a first design of the overall architecture has been drafted. This has been a complex process combining rapid prototyping, requirements identification and architecture definition, in an iterative way.

Some of the identified requirements have been already fulfilled in the Early Prototype, which public release is being presented to the research community during the Early Adopters Workshop, on 1 June 2009, in Crete.

Some other requirements are going to be incorporated in future versions of the LarKC prototype, with the intention to have most of the requirements fulfilled in the last version of the prototype, towards the end of the project.



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	5.3.1	Title	Requirements Analysis and report on lessons learned during prototyping
Work Package	Number	5	Title	The Collider Platform

Date of Delivery	Contractual	M 12	Actual	M 14
Status	version 0.11		final <input type="checkbox"/>	
Nature	prototype <input type="checkbox"/> report <input checked="" type="checkbox"/> dissemination <input type="checkbox"/>			
Dissemination level	public <input checked="" type="checkbox"/> consortium <input type="checkbox"/>			

Authors (Partner)	Michael Witbrock, Blaž Fortuna, Luka Bradeško, CycEur; Mick Kerrigan, Barry Bishop, UIBK; Frank van Harmelen, Annete ten Teije, Eyal Oren, VUA; Vassil Momtchev, Ontotext; Axel Tenschert, Alexey Cheptsov, Sabine Roller, Georgina Gallizo, HLRS			
Responsible Author	Name	Luka Bradeško	E-mail	Luka@cycorp.eu
	Partner	CycEur	Phone	









Abstract (for dissemination)	<p>The main goal of this deliverable is to identify the requirements of the LarKC Platform in terms of the features and capabilities that it must offer, its intended behaviour and interoperability with other LarKC components (specifically the LarKC Plug-ins, to be realised in WP2, 3 and 4) and its intended usage scenarios.</p> <p>After the first fourteen (14) months of the project a wide set of requirements has been identified and a first design of the overall architecture has been drafted. This has been a complex process combining rapid prototyping, requirements identification and architecture definition, in an iterative way.</p> <p>Some of the identified requirements have been already fulfilled in the Early Prototype, which public release is being presented to the research community during the Early Adopters Workshop, on 1 June 2009, in Crete. Some other requirements are going to be incorporated in future versions of the LarKC prototype, with the intention to have most of the requirements fulfilled in the last version of the prototype, towards the end of the project.</p>
Keywords	platform, requirements, architecture, prototyping, lessons learned

Version Log			
Issue Date	Rev. No.	Author	Change
17.1.2009	0.1	Luka Bradeško, Michael Witbrock	Skeleton
09.04.2009	0.2	Georgina Gallizo	First Draft: revision of the skeleton and start collection of



			requirements from WP5 discussions
29.04.2009	0.3	Georgina Gallizo	Analysis of sources, identification and classification of requirements
12.05.2009	0.4	Georgina Gallizo	Input from partners integrated: <ul style="list-style-type: none"> - UIBK: sections 4.1, 4.2 - CycEur: section 4.3 and general sanity check - VUA: section 4.6 - HLRS: sections 4.4, 4.5 Review of requirements sources and priority. New requirements added. Completed chapter 5 on architecture.
15.05.2009	0.5	Vassil Montchev	Input to 2.5.4 section
17.05.2009	0.6	Michael Witbrock	General review and sanity check
18.05.2009	0.7	Michael Witbrock	General review and sanity check update
19.05.2009	0.8	Georgina Gallizo	Update of requirements description in tables
19.05.2009	0.9	Georgina Gallizo	Feedback from WP5 session in Milano
27.05.2009	0.10	Georgina Gallizo	Internal review process by WP5 partners and global review from Michael Witbrock
28.05.2009	0.11	Georgina Gallizo	Comments from Scientific Director (Frank van Harmelen)

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, E-mail: 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 RESPONSABILITA LIMITATA, Milano, Italy, Email: emanuele.dellavalle@cefriel.it
CYCORP, RAZISKOVANJE IN EKSPERIMENTALNI RAZVOJ D.O.O.		Michael Witbrock, CYCORP, RAZISKOVANJE IN EKSPERIMENTALNI 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, E-mail: 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

<p>VRIJE UNIVERSITEIT AMSTERDAM</p>		<p>Prof. Dr. Frank van Harmelen, VRIJE UNIVERSITEIT AMSTERDAM, Amsterdam, Netherlands, Email: Frank.van.Harmelen@cs.vu.nl</p>
<p>THE INTERNATIONAL WIC INSTITUTE, BEIJING UNIVERSITY OF TECHNOLOGY</p>		<p>Prof. Dr. Ning Zhong, THE INTERNATIONAL WIC INSTITUTE, Mabeshi, Japan, Email: zhong@maebashi-it.ac.jp</p>
<p>INTERNATIONAL AGENCY FOR RESEARCH ON CANCER</p>	 <p>International Agency for Research on Cancer Centre International de Recherche sur le Cancer</p>	<p>Dr. Paul Brennan, INTERNATIONAL AGENCY FOR RESEARCH ON CANCER, Lyon, France, Email: brennan@iarc.fr</p>
<p>INFORMATION RETRIEVAL FACILITY</p>		<p>Dr. John Tait INFORMATION RETRIEVAL FACILITY Vienna, Austria Email : john.tait@ir-facility.org</p>



TABLE OF CONTENTS

LIST OF FIGURES	8
LIST OF TABLES	9
LIST OF ACRONYMS.....	10
1. INTRODUCTION	11
1.1. OBJECTIVES.....	11
2. REQUIREMENTS ANALYSIS	11
2.1. PROJECT OBJECTIVES.....	11
2.2. LARKC USE CASES	12
2.3. LARKC PLUG-INS.....	12
2.4. LARKC RAPID PROTOTYPING	13
2.5. THE COLLIDER PLATFORM.....	13
2.5.1. <i>Plug-in Interoperability</i>	14
2.5.2. <i>Support for parallel execution (parallelization “within plug-in”)</i>	14
2.5.3. <i>Support for distributed/remote execution (including parallelization "across plug-ins")</i>	15
2.5.4. <i>Data Access</i>	18
2.5.5. <i>Access to computing resources</i>	19
2.5.6. <i>Support for anytime behaviour</i>	19
2.5.7. <i>Plug-in registration & discovery</i>	20
2.5.8. <i>Monitoring and measurement on plug-ins behaviour</i>	20
2.5.9. <i>Library of support-code for plug-in developers & pipeline constructors</i>	20
3. REQUIREMENTS IDENTIFICATION	21
3.1. LARKC PLATFORM REQUIREMENTS	21
3.2. LARKC PLUG-INS REQUIREMENTS	ERROR! BOOKMARK NOT DEFINED.
4. LESSONS LEARNED.....	26
4.1. API DESIGN.....	26
4.2. TEST API IMPLEMENTATION	27
4.3. TRANSFORMING CYC INTO LARKC PLATFORM	28
4.3.1. <i>Minimizing the code base</i>	28
4.3.2. <i>Interfacing and adding LarKC code</i>	29
4.4. PLUG-INS AND USE CASES IMPLEMENTATION	30
4.5. CODE REPOSITORY AND ISSUE TRACKING	30
4.6. LICENSING ISSUES	31
4.7. MARVIN AND IBIS.....	31
5. FIRST DRAFT OF THE LARKC ARCHITECTURE.....	32
6. CONCLUSIONS.....	33
7. REFERENCES	34



List of Figures

Figure 1: Amdahl's law : efficiency evolution with increasing ratio of serial to parallel parts	17
Figure 2: Symbolic image of the Cyc system and parts used for LarKC	28
Figure 3: Example code coverage analysis for the LarKC platform	29
Figure 4: Platform Architecture – Main components.....	33



List of Tables

Table 1 Platform Features providing incentives to the LarKC user groups	14
Table 2 RDF container types supported by the Data Layer API	18
Table 3 LarKC Platform Resources Requirements	21
Table 4 LarKC Platform Heterogeneity Requirements	22
Table 5 LarKC Platform Usage Requirements.....	22
Table 6 LarKC Platform Interoperability Requirements	23
Table 7 LarKC Platform Parallelization “within plug-ins” Requirements	23
Table 8 LarKC Platform Distributed/remote Execution Requirements	24
Table 9 LarKC Platform Data Layer Requirements.....	24
Table 10 LarKC Platform Data Caching Requirements.....	25
Table 11 LarKC Platform Anytime Behaviour Requirements	25
Table 12 LarKC Platform Plug-in Registration and Discovery Requirements	25
Table 13 LarKC Platform Plug-in Monitoring and Measurement Requirements	25
Table 14 LarKC Platform Support for Developers Requirements	26



List of Acronyms

Acronym	Description
API	Application Programming Interface
DMP	Distributed Memory Parallel
DOM	Document Object Model
DoW	Description of Work
HPC	High Performance Computing
IDE	Integrated Development Environment
LarKC	Large Knowledge Collider
MaRVIN	massive RDF versatile inference network
MPI	Message Passing Interface
SAX	Simple API for XML
SMP	Symmetric multi-processing
SOA	Service Oriented Architecture
VUA	Vrije Universiteit Amsterdam
WP	Work Package
XML	Extensive Markup Language



1. Introduction

1.1. Objectives

The main goal of this deliverable is to identify the requirements of the LarKC Platform in terms of the features and capabilities that it must offer, its intended behaviour and interoperability with other LarKC components (specifically the LarKC Plug-ins, to be realised in WP2, 3 and 4) and its intended usage scenarios.

The document is structured as follows:

- Chapter 2 includes the analysis of the requirements and the different sources that contributed to the collection of requirements
- Chapter 3 makes a classification, by different categories, of the collected requirements from the sources described in the previous chapter and assign concrete identifiers and a brief description to every of them...
- Chapter 4 gives an overview of the lessons learned during the Rapid Prototyping phase (month 1 to month 14 of the project) with regards to requirements and architecture of LarKC
- Chapter 5 offers a first draft of the LarKC Architecture
- Chapter 6 finalises with some conclusions regarding the work performed during the first fourteen months of the project and give some hints about the next steps to be performed with regards to architecture design and prototype implementation

Some of the material in what follows already goes well beyond a mere requirement analysis, and already discusses design options and sometimes even implementation options. In order to maximise the utility of this document, we have not artificially separated these, but are presenting such design and implementation options together with the requirements on which they are based wherever available.

The content of this deliverable will be the basis for the next deliverable within Task 5.3, D5.3.2 Overall LarKC architecture and design v1.

2. Requirements analysis

A variety of sources of requirements have been considered and assessed in order to identify the final set of Requirements that the LarKC Platform must fulfil:

- DoW: Project Objectives, as described in the Description of Work (DoW).
- UC: LarKC Use Cases, as being developed in WP6, WP7a and WP7b
- PLUG: LarKC Plug-ins, as being developed in WP2, WP3 and WP4
- RP: LarKC Rapid Prototyping, performed in WP5 as part of Task 5.2
- CPLAT: LarKC Collider Platform (WP5)

In the following sections a brief description of each of them is given along with, when appropriate, a reference indicating where more information can be found. The source of the requirement will be also referenced in chapter 3 for each of the identified requirements.

2.1. Project Objectives

The DoW outlined the main goals of the LarKC Project, which were necessarily vague at the commencement of a research project. However, over time, these goals have been more concretely defined as the consortium collectively worked to achieve a better understanding of the research goals.

To quote the DoW:



“The Large Knowledge Collider (LarKC): a platform for integrated reasoning and Web-search that scales to zillions of facts”

After analysing the DoW, we tried to answer questions such as the following, which are translated into concrete requirements in chapter 3:

- What is meant by ‘platform’?
- How many is ‘zillions’?
- What hardware architectures does this platform run on?
- Where does this platform execute?
- What is the platform responsible for?
- What other software does the platform interface to?
- How is the platform started?
- How is the platform configured?
- Who uses the platform?
- What is the platform used for?
- What can the platform can do?
- What are the limitations of the platform?
- What is a ‘plug-in’?
- How does the platform interact with a plug-in?
- What is a plug-in responsible for?

2.2. LarKC Use Cases

The three LarKC Use Case Scenarios, being developed in WP6, WP7a and WP7b, aim to validate the LarKC results, including platform and plug-ins. Therefore, they impose a number of requirements that must be considered even during the design phase.

For this reason, a continuous communication between WP5 and the use case WPs has been maintained since the beginning of the project. Additionally, specific joint sessions have taken place in different meetings in order to gather Use Cases requirements for the overall platform design and the Rapid Prototype.

Relevant input from the Use Cases WPs was the detailed “storyboards” elaborated in order to illustrate possible uses of the LarKC Platform. Those storyboards were mapped to concrete LarKC execution steps in deliverable D1.2.1 [2], which presents a possible way of combining different LarKC plug-ins in order to support the implementation of each of the storyboards.

Of special relevance during the Rapid Prototyping phase was the WP6 Urban Computing use case. WP5 and WP6 teams have been collaborating in order to develop the first implementation of the Urban Computing use case. More information about the Urban Computing first implementation can be found in LarKC deliverable D6.3 [18], to be released at the same time as this deliverable D5.3.1.

2.3. LarKC Plug-ins

The LarKC Plug-ins WPs constitute a valuable input to WP5, as some of the components they are developing will be validated through their “plug-ability”, or compatibility, with the LarKC platform. Some of the consortium members involved in the Plug-ins WPs (WP2, WP3 and WP4) are also partners in WP5 and they, directly or indirectly, are driving the inclusion of requirements related to the plug-ins themselves in the discussions of platform structure and function. Other requirements on the platform imposed by plug-ins have been identified through joint discussions between the corresponding WPs.

The main feedback from the plug-ins to the platform, regarding requirements, has been related to:

- API interoperability and usability, including:
 - The interfaces that the plug-ins must support in order to be compatible with the platform



- The data management API used by all LarKC components, but especially plug-ins as it defines the process by which plug-ins and platform consume, process and produce data
- plug-in description of functional and non-functional properties, as an input to the plug-in annotation language.

2.4. LarKC Rapid Prototyping

The goal of building the Rapid Prototype was to function as a first test for the general functionality that the LarKC platform should eventually implement, and to provide as much of that functionality to users as possible, as early as possible. This has enabled the effective collection of requirements, not only for the Rapid Prototype itself, but also for the full platform and the overall LarKC architecture in the longer term. It also allowed platform users (mainly plug-in writers and pipeline constructors) to get familiar with the LarKC APIs and other LarKC platform support features.

The requirements identified as a result of rapid prototyping are collected in the chapter 3, tagged as having source 'RP'. Chapter 4 presents the analysis of the lessons learned during the LarKC rapid prototyping.

2.5. The Collider Platform

In LarKC WP5, "The Collider Platform" requirements were gathered continuously during face-to-face meetings and dedicated workshops, mailing list discussions, and teleconferences, and have led to both the identification of previously unanticipated requirements and the refinement of those from the sources previously mentioned (DoW, use cases, plug-ins and rapid prototyping).

As specified in LarKC deliverable D1.2.1 [2], the use of LarKC involves three categories of users (plug-in developers, pipeline configurators and final users) and comprises the following phases:

- Plug-in construction by plug-in developers
- Pipeline configuration: by pipeline configurators, allowing the combination of existing plug-ins to solve a task
- Platform deployment and execution: by final users

The incentives for each of these three types of users to use LarKC should determine the features the LarKC platform provides, and what can, and should, be left up to the individual users of the various types.

Each "support feature" of LarKC should be justified by at least one of the following goals:

1. For plug-in writers: Make writing of plug-ins easier/more attractive and provide them with the possibility of a wider adoption of their components by the exposure as LarKC plug-ins;
2. For configuration designers (pipeline configurators): To provide an easy-to-build solution by combining plug-ins in a pipeline for a specific task;
3. For end-users: To provide a mechanism for efficient query processing and pipeline execution.

This kind of attractiveness can come from many sources. In the case of developing plug-ins, it may come from ease of re-use of previous code, automated support for transforming a concept to code in an IDE, ease of sharing and obtaining social reinforcement, speed with in which code can be written, clarity and naturalness of the code produced, ease of testing, ease of documentation, and many other sources. For configuration designers, it may come from direct manipulation interfaces, test and metering facilities, ease of reconfiguration, scriptability, etc. For end users, it may come from low latency, high throughput, low cost of execution, appropriateness of results and of number of results, appropriate result visualisation, appropriate metadata about the result-finding process, and so forth.

In furthering support for all classes of user, the following candidates have been discussed extensively in the initial phase of the project as items for which the LarKC platform should provide built-in support:



- plug-in interoperability
- parallel execution (parallelization "within plug-in"): obtaining speedup and scalability
- distributed/remote execution (parallelization "between the plug-ins"): obtaining speedup, avoiding having to move data, obtaining robustness through replication
- data access: obtaining speedup
- data caching
- availability of large-scale computing resources: obtaining speedup; gaining access to LarKC-provided clusters; ease of cloud-deployment (Google App Engine, Amazon EC2, IBM Blue Cloud, Microsoft Azure)
- anytime behaviour
- plug-in registration and discovery
- monitoring/instrumentation
- provision of a library of support code for plug-in builders and plug-in deployers

During the first project phase these features have been analysed with respect to which of them provide incentives to whom amongst the three user-groups. In the table below, these are broken down as features for plug-in-writers, seeking to deploy their code on LarKC (type 1), for configuration designers who are seeking to implement end-to-end operations on LarKC (type 2), and for end-users deploying LarKC as an application (type 3): LarKC support features should generally only be included if they support at least one of [1,2,3].

Table 1 Platform Features providing incentives to the LarKC user groups

Feature	Incentive Group
Interoperability	1, 2
Parallelization	1, 2, 3
Distribution	1, 2, 3
Data Access	1, 2
Data Caching	2, 3
Computing resources	1, 2, 3
Anytime Behaviour	3
Plug-in Registration	1, 2
Instrumentation	1, 2
Code Library	1, 2

In the following sections, each of the abovementioned features is described more in detail.

2.5.1. Plug-in Interoperability

Plug-in interoperability is obtained through:

- The LarKC API and the LarKC data-model, with which all plug-ins and the platform itself must comply;
- The LarKC plug-in description language, including functional and non-functional properties.

2.5.2. Support for parallel execution (parallelization “within plug-in”)

There are two alternatives with regards to the support for parallelization “within plug-in”:

- Leave the parallelization entirely up to the plug-in writer. In this case, there is no requirement imposed on the platform to support “within plug-in” parallelization. There exist several different possible paths to plug-in parallelization, but all current readily usable approaches require preparation for parallelization in the source code structure. No automation or support from the platform for automated code transformation for parallelisation is envisioned. Possible parallelisation mechanisms that may be used within this scenario include:



- OpenMP [14]: code with directives for shared memory architectures
- MPI [15]: message-passing between parallel processes

In this case, the parallel "nature" of the plug-in should be indicated in the plug-in description (provided in the plug-in description language) so that the platform knows that parallelisation is possible and/or likely. The description should also include the necessity (or its lack) to execute the plug-in in a particular environment (e.g. on a cluster with certain interconnection features, or even an individual machine or cluster with a particular identifier).

- Parallelisation support from the platform may be provided under the assumption that plug-in writers adopt a suitable programming model/style (this imposes, therefore, certain rules to the plug-in writers). In this case parallelization "within plug-in" is possible in the sense of having multiple, largely independent instances of the same plug-in. Possible methods for this kind of parallelization are, among others:
 - SATIN [19]: allocate processors depending on recursive call tree
 - MAP-REDUCE : MAP injective $x \rightarrow y$, REDUCE $y \rightarrow \text{many } y \rightarrow z$ (a tutorial of MAP-REDUCE within Hadoop can be found in [20])
 - BOINC [17] ("Thinking@home"): splitting computation in very many independent small-data parts

All of these techniques may be used to run several instances of the same plug-in in parallel and at the same time over different chunks of a split dataset. In this case the LarKC platform should offer the support needed to manage the multiple running instances of the plug-in, the required communication (if any) between them, and to manage generated results. This kind of parallelization can be also seen as a particular case of the distribution model (see section 2.5.3), or "parallelization across plug-ins".

2.5.3. Support for distributed/remote execution (including parallelization "across plug-ins")

A variety of existing approaches have been identified as possible ways to provide support for distributed execution. For some of them is not yet obvious whether they are useful when applied to LarKC-style plug-ins, and this must be further analysed:

- NetKernel [16]: a resource oriented platform providing radical separation of code and data components.
- IBIS [9]: support for remote execution in grid/cluster-like environments. Preliminary support for the utility of this approach has been provided by its use in LarKC for the "MarVIN" prototype experiment in distributed forward inference [11].
- SOA (e.g. implemented via access to remote web services); this placement of computing resources at URLs is part of the NetKernel approach above.
- P2P (many different platforms with many different variations): particular case of distributed computing where participants rely on one another for services.
- BOINC [17], a platform for programming on top of donated computational resources, which may be useful to support loosely coupled LarKC plug-ins for a sort of "Thinking@home".

As long as the executing components (plug-in instances) are independent of one another, they can be distributed so that they run simultaneously and remotely on multiple machines or at multiple network locations. If they are not fully independent, the extent to which this distribution can be achieved is more limited. To support distribution, the platform will be required to include the means to:

- Move an executable to the remote platform (if not already installed), or to location of the most appropriate resources within a platform for its execution
- Move the data to the remote platform, or to the most appropriate resources for its access by the plug-in(s)
- Start the plug-in execution (e.g. by submitting a job to the local scheduler)
- Monitor and control the execution (e.g. check that it is running appropriately, stop execution, re-start execution, etc.)
- Check for intermediate or final results



- Return the results to the caller, or route them for further processing.

Moving data or executables around entails having the means to:

- Check for permissions and handle security issues (authentication, authorization, personal account vs. pool accounts, etc)
- Interoperate with appropriate methods and protocols (web services, VPN, ssh, etc.)

In LarKC, many of these requirements map onto a general requirement to provide appropriate meta-data, either at the level of plug-in type or at the level of a running plug-in instance.

In order to optimize the required processes for distribution, the platform (in case of automated decisions) or the platform administrator (in case of manual configuration) needs advance knowledge about performance of the network (how long it will take to move the data to the execution location and to get them back) as well as about the characteristics of execution on the platform (e.g. how long it takes from submitting a job to obtaining results) and characteristics of the workflow (how often the tasks must be done, in which order,...) to decide whether it makes sense to distribute the execution or not. This information may be given in advance, or, in later versions, it might be learned by techniques such as those under investigation in *WP3 Abstraction and Learning*.

This kind of distributed execution is also known as “task farming”: tasks that are generated by plug-ins are moved to processors based on information about the availability, load, capability, closeness-to-the-data, etc. of the processors. These can be tasks generated by the DECIDE plug-in (i.e. entire plug-ins) or tasks generated inside a plug-in (e.g. because the plug-in-writer wrote the plug-in this way, see section above on *parallelization “within plug-in”*). Such task-farming processes and the corresponding algorithms and policies can be implemented using an existing task-farming platform (e.g. BOINC and IBIS are both such platforms, geared towards different compute-environments). In any case, the LarKC platform must offer support for the deployment and execution of the task-farming environment. At the time of writing this deliverable, task-farming using IBIS is being experimented with for possible later inclusion in the platform.

The simpler "distributed pipeline" approach (where each plug-in is allocated to a single (fixed) machine), can be implemented as a special case of task farming. Similarly, plug-ins that are just wrappers around a call to an external service (e.g. the current Sindice-IDENTIFY) can be also be handled as a special case, by farming out just a wrapper that makes the service-call. In these degenerate cases, farming may not be particularly efficient. Such "wrapper-for-webservice-calls" plug-ins must be done if the code is only remotely accessible or the data is not allowed to be shipped or if the amount of data to be shipped is high such that the time for shipping the data is much longer than the time for executing the task. Farming is most efficient if significant work must be done on large amounts of local data, e.g. in a cluster-environment.

The efficiency of a task farming algorithm depends on several factors, namely:

- the number of processes available, i.e. number of tasks that can be executed at the same time;
- efficiency of the code execution on a given process, i.e. hardware type;
- the time for shipping the data, which depends on
 - the amount of data that has to be shipped between the task executing processes,
 - the access patterns to access the data, i.e. how often is a small / medium / large piece of data accessed,
 - the bandwidth of the network between the processes, i.e. the time for actually communicating the data,
 - the latency of the network, i.e. the time for setting up the communication.

The communication time $C(N,p)$ is a function of $M(N,p)$ (amount of data that have to be communicated), L (latency in the communication), BW (Bandwidth): $C(N,p) = L + M(N,p) / BW$.

Remark: Processes (p) and network (N) are to be understood as being possibly on very different levels. Processes might be cores in a CPU, nodes in a cluster, a set of clusters at different sites, or



individual PCs in a thinking@home-like environment. The networks between them might be the BUS within a CPU, the high-speed network within a cluster, a scientific high-bandwidth wide area network like GEANT [22], or the internet in its whole heterogeneity. The task farming algorithm needs to have a scheduling algorithm to assign a task to a process in an optimized way such that the highest throughput is achieved for executing the entire task farm, not a single task. This scheduling algorithm should be topology-aware, i.e. know about the heterogeneity of processes and network. It also needs to know properties of the codes with regards to underlying hardware.

The efficiency of any parallel algorithm is measured by the ratio of speed-up to number of processes used. I.e. denote with $T(N,1)$ the time for a serial execution of a task farm of size N , with $T(N,p)$ the time for executing the same size farm on p processes. Then the speed-up $S = T(N,1) / T(N,p)$, and the efficiency $E = S / p$. While in a perfectly parallel algorithm, the efficiency is 100%, it is in practise limited by a part of the execution which cannot be done in parallel. Sequential parts of a program typically are:

- Collecting information (maximum over all processes)
- Reading/writing data
- Task Decomposition
- Setting up communication patterns

The efficiency is decreasing with increasing ratio of serial to parallel parts, and is getting worse with increasing numbers of processors. This is known as Amdahl's law [21].

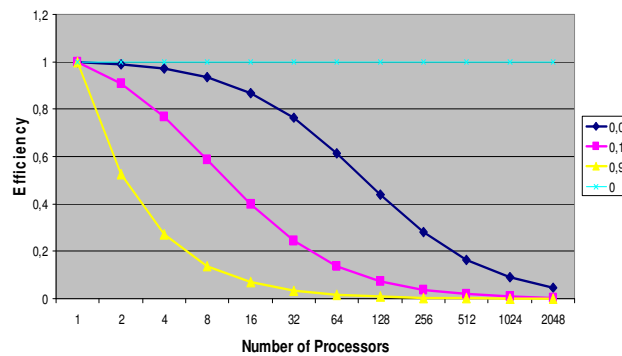


Figure 1: Amdahl's law : efficiency evolution with increasing ratio of serial to parallel parts

In general, two types of task farming can be distinguished:

- black-box farming: the allocated tasks are indivisible and not inspectable by the platform. In this case the plug-in developer must do the splitting himself, but can farm subtasks back out to the farming system (the same farming system that was used to allocate the original task of a plug-in).
- white-box farming: the platform inspects the task and tries to split it up and farm it out. For LarKC, white-box farming is more desirable, and wherever possible plug-ins should be designed with this in mind.

The choice of whether to perform remote/distributed execution and which kind of distribution to perform is influenced by, *inter alia*:

- the volume of data that must be transported
- network features between the remote locations (bandwidth, latency,...)
- frequency of remote calls
- dependency / independency of operations within / between plug-ins

The choice for or against any of these paradigms will also partially determine the kind of hardware on which a particular LarKC workflow (pipeline) can run efficiently (SMP, DMP, hybrid, high/low bandwidth, etc. For more details, see D5.1 [4]).



2.5.4. Data Access

The LarKC platform must provide a common model for accessing data. Ideally, this model should abstract from local or remote access, and from passing data physically or via a pointer. For this purpose, a Data Layer API has been defined and will be refined and improved according to the needs of the LarKC architecture, as the project progresses.

The Data Layer API is offered as part of the LarKC platform and used by the LarKC plug-ins to persistently store and exchange information. Thus, an important requirement is efficient storage and access to data and the related meta-data. The knowledge representation formalism adopted by LarKC, described in D1.1.3 [6], uses an extended RDF-based data model with flexible support for named graphs and triple-sets. An important distinctive feature of the LarKC Data Layer is the use of triple-sets as a primitive, permitting arbitrary sets of RDF statements to be passed by reference. A scenario is possible in which plug-ins share large volumes of data, passing this data from one to another via a shared data layer. Technically, one plug-in can pass a reference to a triple-set (a subset of the data in a dataset stored in a shared repository), which is then subject to processing at the next step of the pipeline [2].

In the following paragraphs, some details on design and implementation of the current Data Layer API are given.

The current Data Layer API is built on the ORDI API [7] and reuses the Sesame open source framework classes for managing RDF data [8]. The data layer supports but is not limited to the following major types of serializable RDF data:

- SetOfStatement – the most generic RDF type that is capable only to iterate triple or quadruples (extended with graph name) statements
- RdfGraph – a collection of quadruple statements to represent a named graph
- Dataset – a set of RDF graphs associated with SPARQL endpoint to process queries
- LabelledGroupOfStatements – a group that could associate arbitrary RDF statement with URI label; the labelled group of statement has different semantics from the named graph and it is aimed to achieve far superior performance in contextualizing RDF data

The following table explains the different implementation of the support RDF type. Next to each implementation is described the mechanism to pass data.

Table 2 RDF container types supported by the Data Layer API

Class	Description	Usage	Cost to pass 10 ⁶ statements
SetOfStatement Impl	RDF statements passed by value	Services to return RDF data (e.g., SPARQL construct and describe queries)	350MB memory
RdfGraphInMemory	Named graph passed by value	Pass named graphs between plug-ins	350MB memory
HTTPRemoteGraph	RDF statements published on the Web passed by reference using a URL	Load chunks of RDF from URL	few KBs + time to load the data
RdfGraphDataSet	Named graph passed by reference	Pass named graphs using a SPARQL endpoint	few KBs + time to load the data from a SPARQL endpoint
DataSetImpl	SPARQL dataset passed by reference (the statements are selected based on the graph name matching)	Constrain RDF data exposed by a SPARQL endpoint with a list of graph names	few KBs + time to load the data from a SPARQL endpoint



Class	Description	Usage	Cost to pass 10 ⁶ statements
LabelledGroup Of StatementsImpl	RDF statements associated to a group identified by a tripleset [6]	Pass arbitrary sets of RDF statements between plug-ins (e.g., graph priming)	few KBs + time to associate the statements + time to read with a wildcard

The Data Layer API should be considered as a passive functional service. It is responsibility of the individual plug-in, possibly under the direction of a DECIDE plug-in, to efficiently utilize data layer infrastructure.

The platform will need to support data caching, and could do so at several possible levels:

1. Data layer – the most frequently or likely pages of RDF data to be loaded in the memory
2. Pipeline or plug-in level – there is a run-time trade-off between different options:
 - a. always leave data remote
 - b. make a full local copy (can be decided off-line)
 - c. caching / data-warming during computation (must be decided at run-time)

While useful, the data caching mechanisms could also significantly increase the system complexity and introduce major synchronization overhead especially in distributed environment. In the first prototype version and until we are able to accumulate a significant usage history, the design has focussed only on providing a low level, generic data caching mechanism at the data layer level. For the future releases more sophisticated data caching at the pipeline or plug-in level will be considered.

2.5.5. Access to computing resources

Certain plug-ins or data may require access to large computing or storage resources. In general, it is not the responsibility of the LarKC platform to provide the availability of such resources, but it must be the plug-in provider who ultimately determines whether a plug-in requires special resources where to be executed. The plug-in description must therefore include enough information for the platform to know, or to decide, where the plug-in must be executed. The plug-ins which requires some special resources or special location where to be executed must be annotated properly with the plug-in annotation language and should provide enough information to the platform to be able to present this information via the Plug-in Registry.

It is being considered that the LarKC platform offer a “catalog” of the supported resources so that the plug-in provider can choose among them. The platform should offer support for execution in different kind of computational resources, such as a cluster (via the IBIS platform, via another middleware,...), in cloud computing resources via a cloud API, etc. In the cases where the platform does not support access to the required resources by certain plug-ins, either the execution of those plug-ins will not be supported or an alternative resource support may be offered.

2.5.6. Support for anytime behaviour

Even in our earliest experimentation with LarKC pipelines, it has emerged that a desired behaviour of the LarKC platform is that the “user” (an actual user, or another plug-in) could terminate a search or reasoning task at any time and still have received some meaningful answer (or set of results). Such behaviour is described by the term “anytime”. The principal advantage is that it allows one to trade response time off against accuracy or quantity of results, as appropriate for the processing task.

Furthermore, a LarKC platform is constructed from many components that work together (under the guidance of a DECIDE plug-in) that together deliver results to the user. These components will in turn need to communicate and will likely also pass “intermediate results” between themselves.

Therefore two aspects of anytime behaviour should be supported by the platform:

- Between the end-user and the LarKC Platform
- Between the plug-ins or between plug-ins and the LarKC Platform



Regarding the communication between the user and the platform, the platform acts to the outside world as a standard SPARQL endpoint as defined by W3C, which in fact originally does not return results in the anytime manner. For this reason the server part should be modified to support this behaviour in a way that is still compliant with the SPARQL protocol. The modifications should include some internal changes on the code where server calls the platform and also switching to the SAX XML parser instead of using DOM which enables us to easily implement streaming.

2.5.7. Plug-in registration & discovery

The LarKC platform must allow the registration of new plug-ins located either locally or at remote sites. There must be a process of registration or announcement that the plug-in is available and under which conditions. The plug-in registration process must therefore include the description of the plug-in from a functional and a non-functional viewpoint, expressed in the so called “Plug-in Annotation Language”. For more information about functional and non-functional features, and the ontology representing them, see [3].

The registration of plug-ins may be done by explicit registration in a single repository or by crawling other possible plug-in repositories. The explicit registration of plug-ins may be even done at run time. The crawler approach could be used as part of an offline process that would feed the Plug-in Registry with the necessary information for a subsequent discovery.

The plug-in description included in the Plug-in Registry during the registration process will allow to discover plug-ins based on their particular properties and expected behaviour. To handle this, a Service Discovery mechanism is needed. Such a mechanism should be available both to DECIDE plug-ins and to pipeline configurators (human user, type of user 2, according to section 2.5)

2.5.8. Monitoring and measurement on plug-ins behaviour

The platform should provide a mechanism for monitoring and measurement of plug-in behaviour, allowing users, and decide plug-ins, to get performance information such as:

- memory use
- CPU use
- patterns of data access
 - volumes
 - frequency
 - grain size (only access to entire data-set or per data-item)

And other kind of information (for each plug-in and pipeline), so that performance may be anticipated better in the subsequent runs, such as:

- the executed query
- size of the input data
- size of the output data

This is important for the role of LarKC as an experimentation platform. This information can be used to drive the analysis of plug-ins non-functional properties to support a more accurate description of plug-in requirements and expected behaviour and for pipeline optimization purposes.

2.5.9. Library of support-code for plug-in developers & pipeline constructors

A supporting library of components’ templates and wizards, as well as the corresponding documentation, may be provided to plug-in developers and pipeline configurators in order to ease the development and deployment of LarKC components. Some possibilities are:

- wrappers to existing interface standards (e.g. DIG reasoner) according to the LarKC plug-in API
- templates of the plug-ins with all the code structure annotation files and example data passing (a particular case of this is the templates for DECIDE plug-ins, such as “scripted DECIDE” or “intelligent DECIDE”)



- wizards which will be able to generate the specific plug-in source code and annotation files and maybe even a working pipeline example
- small integrated development environment as the Eclipse plug-in which will consists of code generation wizards, project maker and a small integrated SPARQL client, so users will be able to do all the testing and developing from one place

3. Requirements Identification

In this chapter requirements are identified and described, classified in the following groups:

- LarKC Platform Requirements, tagged as PLAT_S_nn, where S is the subgroup and nn is an identification number
- LarKC Plug-ins Requirements, tagged as PLUG_nn
- Data Model Requirements, tagged as DM_nn
- Storage Requirements, tagged as ST_nn

For every group, a table is given with the following information:

- ID: Requirement identifier, consisting on a tag corresponding to the group plus an identification number (e.g. PLAT_R_01 for first identified Platform Requirement related to Execution Resources)
- Requirement: Short description of the requirement
- Source: Origin of the requirement, as described in chapter 2. Main source(s) is mentioned here, although other(s) may have also contributed, in a minor extent, to the identification of the requirement.
- Priority: Indication of the level of importance of the requirement. It may contain the following values:
 - M: Mandatory. For those requirements that LarKC MUST fulfil.
 - O: Optional. For those requirements that LarKC MAY include as desirable, but not mandatory.
 - NR: Non-required. For those requirements that are not required at all
- An extended description of some requirements is given, when necessary.

3.1. LarKC Platform Requirements

The LarKC platform requirements have been classified different in groups and included in the following tables:

Table 3 LarKC Platform Resources Requirements

Resources (PLAT_R_nn) (computing / storage)			
ID	Requirement	Source	Priority
PLAT_R_01	Distributed reasoning platform built on a high-performance computing cluster	DoW, page 5	O
The platform may take advantage of high-performance computing resources use (i.e. a ‘massively’ parallel infrastructure of many processors, cluster of SMP nodes) in order to improve scalability and achieve the scalable reasoning paradigm. This does not mean that the complete platform must be deployed on a HPC cluster. It must be analysed how to perform distribution of platform support components and plug-ins and which of the HPC systems fits the current user configuration best.			
PLAT_R_02	Distributed reasoning platform built via “computing at home”	DoW, page 5	O
The platform may take advantage of desktop grids and volunteer computing systems (i.e. computing@home) which utilize the free resources available in Intranet and Internet environments for supporting large-scale computation and storage. Similarly to PLAT_R_01, the optimal redistribution model among the available resources is to be elaborated			
PLAT_R_03	Support for accessing computing resources	CPLAT	O



Resources (PLAT_R_nn) (computing / storage)

ID	Requirement	Source	Priority
	The platform may offer support for accessing special resources (e.g. executing a certain plug-in in a cluster). This support may be offered as part of the platform functionality OR as special add-ons that will be booted only when needed.		

Table 4 LarKC Platform Heterogeneity Requirements

Heterogeneity (PLAT_H_nn)

ID	Requirement	Source	Priority
PLAT_H_01	Massive inference is achieved by distributing problems across heterogeneous computing resources and coordinated by the LarKC platform.	DoW, page 9	M
	The LarKC platform must support the management (instantiate, monitor and control) of all distributed components running in heterogeneous and possibly remote resources. They can be both platform-dependant components and plug-ins.		
PLAT_H_02	The platform shall comprise a pluggable architecture that will ensure that computational methods from different fields can be coherently integrated.	DoW, page 7, 8	M
	Computational methods from different fields (cognitive science (human heuristics), economics (limited rationality and cost/benefit trade-offs), information retrieval (recall/precision trade-offs), and databases (very large datasets)) will be addressed within the different the plug-ins. The platform architecture must allow those plug-ins to be plugged into the platform and executed in combination with each other in a coherent way.		
PLAT_H_03	Instead of being built only on logic, the Large Knowledge Collider will exploit a large variety of methods from other fields: cognitive science (human heuristics), economics (limited rationality and cost/benefit trade-offs), information retrieval (recall/precision trade-offs), and databases (very large datasets)	DoW, page 7, 8	M
	LarKC must support different kind of plug-ins, implementing methods from different fields.		

Table 5 LarKC Platform Usage Requirements

Usage Requirements (PLAT_U_nn)

ID	Requirement	Source	Priority
PLAT_U_01	Plug-in construction by plug-in developers	CPLAT	M
	The platform must allow integration of plug-ins written by a variety of plug-in developers, who may be LarKC partners or external contributors		
PLAT_U_02	Pipeline configuration by pipeline constructors	CPLAT	M
	The platform must allow the configuration of multiple pipelines, by pipeline configurators, combining existing plug-ins to solve a task. This configuration will usually consist mainly of the development of a DECIDE plug-in		
PLAT_U_03	Platform deployment and execution by end-users	CPLAT	M
	The platform must allow deployment and execution by end-users seeking an answer to a given query, without their needing to care about the internal configuration of the LarKC components		
PLAT_U_04	Platform deployment in the end-user machine	CPLAT, RP	O
	The platform may allow for local deployment and execution, possibly as multiple running processes, and possibly in diminished form, on a desktop machine, where such execution supports the envisioned pipeline workflow.		



Usage Requirements (PLAT_U_nn)			
ID	Requirement	Source	Priority
PLAT_U_05	Access to the remotely deployed platform from the end-user machine	CPLAT, RP	M
The platform must allow deployment as a remote server and its invocation from end-user machines			
PLAT_U_06	Multi-user platform	CPLAT, RP	M
A Platform instance running remotely must allow concurrent access by multiple users.			
PLAT_U_07	Low platform overhead	UC	M
The Platform must permit execution of queries in a “time-constrained” interactive way, without unreasonable platform-imposed latency. For certain use cases, plug-ins may support rapid response, and a user who introduces a query may require an answer in a reasonable time (“soft real-time” requirements)			
PLAT_U_08	Offline execution	UC	M
The Platform must permit execution in an offline (or batch) mode. For certain use cases, there is no need to get an immediate answer to a given query, but the pipeline may be executed in a batch mode without any time limit constraint.			
PLAT_U_09	Quality specification by users	UC	M
The Platform must offer a way for the user to specify the desired quality or other attributes of the answer, as part of the input parameters (E.g. max time of response, min n° answers,...).			

Table 6 LarKC Platform Interoperability Requirements

Support for plug-in interoperability (PLAT_I_nn)			
ID	Requirement	Source	Priority
PLAT_I_01	Common LarKC API	CPLAT, PLUG	M
The LarKC platform must offer a common API for the interoperability between platform and plug-ins and between plug-ins. This API offers an interface for the different kind of plug-ins that are compatible with the LarKC platform.			
PLAT_I_02	Common LarKC Data Layer API	CPLAT	M
The LarKC platform must offer a common Data Layer API for the uniform access to the data by plug-ins and platform.			
PLAT_I_03	Common LarKC plug-in description language	CPLAT, PLUG	M
The LarKC platform must offer a common plug-in description language, which both platform and plug-ins can understand for their interaction between each other.			

Table 7 LarKC Platform Parallelization “within plug-ins” Requirements

Support for parallelization “within plug-in” (PLAT_P_nn)			
ID	Requirement	Source	Priority
PLAT_P_01	Platform support for parallelization “within plug-ins”	CPLAT	NR
“Within plug-in”parallelization code must be designed by the plug-in writer at the time of development of the plug-in. There exist several different possible paths to plug-in parallelization, but all current readily usable approaches require preparation for parallelization in the source code structure. No automation or support from the platform for automated code transformation for parallelisation is envisioned.			
PLAT_P_02	Platform support for running multiple instances of one plug-in	CPLAT	M



Support for parallelization “within plug-in” (PLAT_P_nn)

ID	Requirement	Source	Priority
	Support for parallelization “within plug-in” is currently envisioned in the sense of having multiple independent instances of the same plug-in type. The LarKC platform should offer the support to manage different running instances of the plug-in, the communication (if any) between them and to manage generated results		

Table 8 LarKC Platform Distributed/remote Execution Requirements

Support for distributed/remote execution (parallelization "across plug-ins") (PLAT_D_nn)

ID	Requirement	Source	Priority
PLAT_D_01	Migrate executable plug-ins	CPLAT	O
	The platform may offer support to migrate executable plug-ins to the location of the most appropriate resources for its execution (this may be the platform location itself or another resource location, such as e.g. a computing cluster)		
PLAT_D_02	Migrate data	CPLAT	O
	The platform may offer support to move data to the location of the most appropriate resources for its access by the plug-in(s) (this may be e.g. the plug-in executable location, in order to allow for local access to data)		
PLAT_D_03	Start plug-in execution	CPLAT	M
	The platform must offer support to start plug-in execution, independently of whether the plug-in is located in the same physical resources as the platform or in a remote location.		
PLAT_D_04	Support to monitor and control plug-in execution	CPLAT	M
	The platform must offer support for monitoring and controlling plug-in execution (e.g. to check that it is running appropriately, stop execution, re-start execution, etc), independently of whether the plug-in is located in the same physical resources as the platform or in a remote location. The platform itself is not expected to control the plug-ins, it simply offers the support to do so. Actual control actions are to be triggered by one or more DECIDE plug-ins.		
PLAT_D_05	Check for intermediate or final results	CPLAT	M
	The platform must offer support to check for the presence of intermediate or final results generated by a plug-in.		
PLAT_D_06	Get the results back and transfer them	CPLAT	M
	The platform must offer support to retrieve plug-in results and/or move them to another location		
PLAT_D_07	Security issues	CPLAT	O
	The platform may offer support for secure movement of data and executables to remote resources. In some cases, those remote resources may require an additional security process (authentication, authorization, certificate provision, etc)		

Table 9 LarKC Platform Data Layer Requirements

Data Access (PLAT_DL_nn)

ID	Requirement	Source	Priority
PLAT_DL_01	Data Layer API	CPLAT	M
	Platform must provide, as part of the supporting tools, a Data Layer API as a common model for accessing and managing data by the different plug-ins and by the platform itself.		
PLAT_DL_02	Efficient support of LarKC data model	RP	M
	Data Layer API implementation must efficiently support the LarKC data model.		
PLAT_DL_03	Transparent access to remote RDF data	RP	M
	Data Layer API must support transparent to remote RDF data published as remote SPARQL endpoint or HTTP page.		



Table 10 LarKC Platform Data Caching Requirements

Data Caching (PLAT_DC_nn)			
ID	Requirement	Source	Priority
PLAT_DC_01	The most frequently used pages of RDF data should be stored in the memory in order to provide a faster access time.	CPLAT,	M
PLAT_DC_02	Data warming/cooling	CPLAT	O
This concerns support to avoid repeated remote access of the same data, as well as predicting which data will have to be accessed next, and moving this data closer to the computation (data warming/cooling). This requirement should be further investigated.			

Table 11 LarKC Platform Anytime Behaviour Requirements

Anytime Behaviour (PLAT_AB_nn)			
ID	Requirement	Source	Priority
PLAT_AB_01	Anytime Behaviour between the end-user and the LarKC Platform	CPLAT, RP	M
The end-user should be able to get intermediate (set of) results and terminate the execution of the platform at anytime. The behaviour required from the LarKC platform and supporting plug-ins should, wherever possible, start to return query results as soon as they become available with the intention that the longer ones wait, the more results are produced and the quality of results improves.			
PLAT_AB_02	Anytime Behaviour between the plug-ins or between plug-ins and LarKC Platform	CPLAT	O
Certain plug-ins in the pipeline may start producing results, and pass them as input to the next plug-in in the pipeline, as soon as they become available, without the need to have the complete answer.			

Table 12 LarKC Platform Plug-in Registration and Discovery Requirements

Plug-in registration and discovery (PLAT_PRD_nn)			
ID	Requirement	Source	Priority
PLAT_PRD_01	Registration of plug-ins	CPLAT	M
The LarKC platform must allow the registration of new plug-ins located either locally or at remote sites. This registration should be able to happen at start-up, or dynamically during platform execution.			
PLAT_PRD_02	Plug-in Registry maintenance	CPLAT	M
The platform must maintain a plug-in registry with all information related to registered plug-ins: including functional and non-functional features. Particular attention will be required to ensure that information about the platform and plug-ins is accessible to DECIDE plug-ins			
PLAT_PRD_03	Crawling of plug-ins	CPLAT	O
A crawling process may take place offline in order look for new plug-ins and to feed the plug-in registry with newly available plug-ins			
PLAT_PRD_04	Service Discovery mechanism	CPLAT	M
The platform must offer a Service Discovery mechanism which allows for searching across the plug-ins in the plug-in registry. This may be available both human and automated pipeline configurators (SW components, especially DECIDE components, or human persons).			

Table 13 LarKC Platform Plug-in Monitoring and Measurement Requirements

Monitoring and measurement on plug-ins behaviour (PLAT_M_nn)			
ID	Requirement	Source	Priority
PLAT_M_01	Behaviour measurement mechanism	CPLAT, PLUG	O



Monitoring and measurement on plug-ins behaviour (PLAT_M_nn)

ID	Requirement	Source	Priority
	The LarKC platform may offer a mechanism to measure some performance parameters of the plug-ins during run-time		
PLAT_M_02	Performance log (Monitoring logging)library	CPLAT, PLUG	O
	The LarKC platform may offer a mechanism to keep and access a performance library where performance of plug-ins previously executed is stored. This would allow DECIDE plug-ins and pipeline (human) configurators to choose the most appropriate plug-in based on reliable historic information.		
PLAT_M_03	Benchmarking of plug-ins	CPLAT, PLUG	O
	The LarKC platform may offer a mechanism to benchmark plug-ins prior to their real execution. This would allow plug-in developers to estimate the plug-in performance in order to include more accurate information as part of the plug-in description non-functional parameters.		

Table 14 LarKC Platform Support for Developers Requirements

Support to plug-in developers and pipeline constructors (PLAT_S_nn)			
ID	Requirement	Source	Priority
PLAT_S_01	Support code library	CPLAT	M
	The LarKC platform must offer support for plug-in developers and pipeline constructors in the form of components templates and / or code libraries		

3.2. LarKC Plug-ins Requirements

Table 15 LarKC Plug-ins Requirements

ID	Requirement	Source	Priority
PLUG_01	LarKC API compliance	RP, CPLAT	M
	All plug-ins must be compliant with the LarKC API		
PLUG_02	LarKC Data Layer API compliance	RP, CPLAT	M
	When interoperability between different plug-ins that exchange RDF data is required then they should use the LarKC Data Layer API.		
PLUG_03	Plug-in description	RP, CPLAT, PLUG	M
	All plug-ins must be accompanied by a plug-in description using the plug-in annotation language, including functional and non-functional parameters		

4. Lessons Learned

4.1. API Design

The initial design of the LarKC API was sketched out during the workshop in Ljubljiana in August 2008. During this workshop it was decided to change the names of the different plug-in types, in order to better reflect their nature:

Retrieve => Identify



Abstract => Transform
Select – no change
Infer => Reason
Decide – no change

During the implementation of the first prototype it was realised that there are essentially two types of transform components in a pipeline. The first prototype pipeline used the Sindice [10] Web service to ‘identify’ RDF resources on the Web that could be used to answer the input SPARQL query. However, the sindice service comes in two forms – triple pattern search and keyword search – neither of which can use the input SPARQL query directly. Indeed similar services such as SWOOGLE and Watson also use a variety of input data forms. Hence it became clear that a transformation of the input SPARQL query is required and to facilitate this, a new plug-in interface was created, ‘QueryTransformer’, as a special case of TRANSFORM plug-in.

Originally, it was planned for plug-ins to accept and return certain data structures that were identified from the proposed LarKC data model. For example, it made sense that a SELECT plug-in would accept a collection of RDF graphs (data-set) and return a subset of these triples (triple-set). However, this approach meant that it became impossible to wire together two select components in series in a pipeline without significant extra programming. So after several revisions of the API, it was realised that from a plug-in’s point of view, the type of the data structures used as input was not relevant. The plug-in just needs to be able to access and process the triples. Therefore, the plug-in interfaces were modified to accept and return only the most abstract data structures containing RDF triples, thus imposing less restrictions on how plug-ins are assembled in a pipeline and giving plug-in writers greater freedom to return RDF triples in data-structures appropriate for the algorithm encapsulated within their plug-in. Ensuring compatibility between plug-ins will be done by the DECIDER plug-ins and/or pipeline configurators, based on plug-ins metadata (plug-ins description through plug-in annotation language).

4.2. Test API implementation

The first test API implementation served to validate the very first definition of the API, V0.0.1. This allowed us to get an initial feedback on the appropriateness of the API for fulfilling the LarKC requirements identified until that moment.

The requirement for anytime behaviour, i.e. the ability of the platform to provide answers over time of ever increasing quality/quantity, required some changes in the behaviour of the pipeline. In this case, some negotiation is necessary so that plug-ins pass data along the pipeline in some kind of agreed fashion. For this purpose, an abstract ‘Contract’ data structure is passed to each plug-in when their services are invoked. This contract allows the decider (or the next plug-in in sequence for the case of simple deciders) to provide some meta-data about the invocation, e.g. the amount of data required.

In order to keep plug-in interfaces simple and to reduce implementation complexity, anytime behaviour has been achieved by wrapping each plug-in in its own threaded container and connecting containers together using asynchronous queues. Such a modification lends itself nicely to future work on distributing plug-ins over a network, where remote invocation and parameter marshalling could be achieved using more intelligent containers.

Lastly, distributed execution brings the implication that plug-in instances have a longer lifetime than the query execution task that requires them. It also brings the possibility that plug-ins may be involved in several pipeline executions simultaneously. Therefore plug-ins should, wherever possible, be essentially stateless and have an abstract ‘Context’ that the decider can pass each time a plug-in is invoked. For example, the context for a (stateless) Sindice identifier component might contain information about what pages of results were returned the last time it was invoked. This would allow the plug-in to carry on execution from the previous state relevant for the pipeline. Persistence should be managed via the Data Layer.

4.3. Transforming Cyc into LarKC platform

Cyc, whose inference engine structure was offered to LarKC as the basis of the platform, is itself a reasoning platform with a huge (more than 1150 classes) code base; not every element of this code base was needed for the initial platform prototype LarKC.

In order to be used by LarKC developers, and in order not to release unnecessary parts of Cycorp's IP, the Cyc code had to be minimized to retain only the needed features and then reorganized and interfaced according to the LarKC API. Therefore the transformation was divided into a two steps - minimization and reorganization.

4.3.1. Minimizing the code base

During the earliest stage of the project, Cycorp Europe identified parts of the Cyc system which were suitable and necessary for the LarKC platform. At first the process was done by hand and was a time consuming and uncertain task. It soon became clear that this approach could lead to a variety of problems:

- LarKC was still in its early design stage and it was not known what requirements it would have that could be satisfied by Cyc code
- The LarKC code and requirements were changing constantly, as bugs were fixed and various plug-in and platform ideas were tried
- The Cyc source code was changing, as new features were introduced, and also due to bug fixes being performed for reasons both related and unrelated to LarKC.

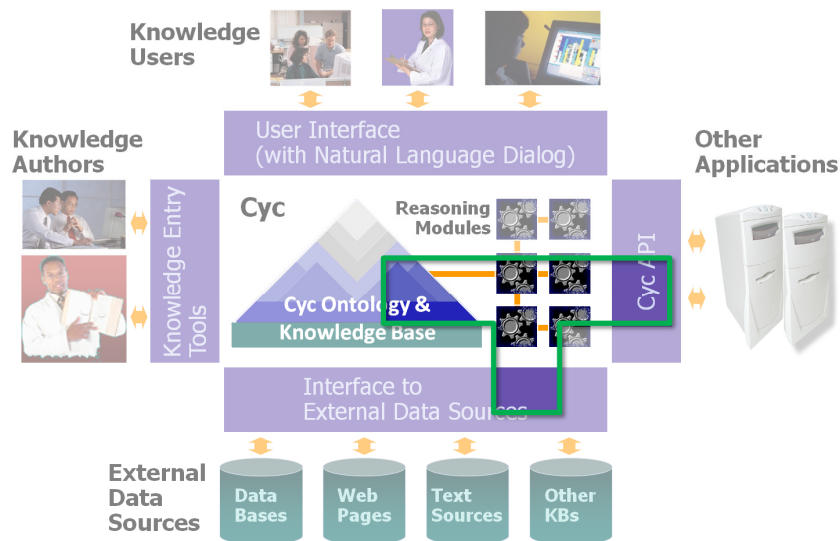


Figure 2: Symbolic image of the Cyc system and parts used for LarKC

In overcoming the problems stated above, Cycorp Europe initiated a somewhat novel approach to managing and identifying the needed parts of the code.

In order to make this process as economical as possible and independent of changes on both the LarKC and Cyc sides of the code, code reduction was automated using the Code coverage tool EclEmma (<http://www.eclEmma.org/>) and an additional Eclipse plug-in, written by Cycorp Europe, which removed uncovered methods and classes. Using this Eclipse plug-in, all of the removed methods and classes were saved externally, identified by a numeric ID, and all of the calls to those classes and methods were replaced by an exception, identifying the missing method.



For the initial code coverage analysis the existing LarKC code was used together with unit tests addressing the planned features of the platform. Additionally, explicit steps were taken to ensure the retention of methods known to be important, but not yet covered by tests.

Element	Coverage	Covered Instructio...	Total Instructions
src	65,4 %	622588	952550
com.cyc.cycjava.cycl	74,5 %	380346	510845
com.cyc.cycjava.cycl.cyc_testing	84,5 %	8022	9492
com.cyc.cycjava.cycl.cyc_testing.kb_cont	99,1 %	452	456
com.cyc.cycjava.cycl.inference	72,8 %	17202	23632
com.cyc.cycjava.cycl.inference.harness	66,2 %	86838	131240
com.cyc.cycjava.cycl.inference.modules	73,2 %	11698	15970
com.cyc.cycjava.cycl.inference.modules.r	70,2 %	31586	45023
com.cyc.cycjava.cycl.owl	97,2 %	968	996
com.cyc.cycjava.cycl.sbhl	67,6 %	32828	48557
com.cyc.cycjava.cycl.sksi.sksi_infrastruct	100,0 %	59	59
com.cyc.tool.subl.jrtl.nativeCode.subLisp	25,8 %	13287	51599
com.cyc.tool.subl.jrtl.nativeCode.type.coi	27,2 %	5837	21432
com.cyc.tool.subl.jrtl.nativeCode.type.exc	2,5 %	41	1634
com.cyc.tool.subl.jrtl.nativeCode.type.nu	21,2 %	707	3339
com.cyc.tool.subl.jrtl.nativeCode.type.op	37,1 %	1190	3209
com.cyc.tool.subl.jrtl.nativeCode.type.str	28,1 %	1683	5993
com.cyc.tool.subl.jrtl.nativeCode.type.syr	67,5 %	4751	7036
com.cyc.tool.subl.jrtl.translatedCode.subl	47,3 %	21031	44453
com.cyc.tool.subl.parser	0,0 %	0	13721
com.cyc.tool.subl.ui	0,0 %	0	1500
com.cyc.tool.subl.util	31,7 %	1186	3740
eu.larkc.core	74,3 %	248	334
eu.larkc.core.config	0,0 %	0	40
eu.larkc.core.data	31,4 %	268	853
eu.larkc.core.data.iterator	47,2 %	83	176
eu.larkc.core.data.util	10,0 %	28	280
eu.larkc.core.metadata	54,1 %	333	616
eu.larkc.core.orchestrator	74,1 %	315	425
eu.larkc.core.orchestrator.servers	21,6 %	674	3126
eu.larkc.core.pluginManager	58,0 %	29	50

Figure 3: Example code coverage analysis for the LarKC platform

With this approach it is possible to easily revert and add the missing methods, when some change in LarKC entails the use of an additional Cyc feature which was not included before. For smaller features, the time needed to include the missing feature is , from the reference Cyc code base, is measured in minutes. For more substantial design changes to LarKC, code coverage and removal has to be repeated, which takes around 1 to 2 working days.

The frequency with which it was necessary to add and remove Cyc code used by LarKC was very high at the beginning of the prototype design and construction, but is close to zero now that the platform prototype is achieving some stability in features and functionality

4.3.2. Interfacing and adding LarKC code

The result of the source code minimization step is a platform with around 560 classes, which are divided into support code, knowledge base code, inference engine code and the new LarKC platform code (API and functionality).



Because the original Cyc source code had been automatically translated to Java from Lisp¹, it was not as natural as it might have been for some developers outside Cycorp Europe to manipulate and call the native functions provided by the platform. For this reason all of the essential functions were wrapped by an API, which reflects the need of the LarKC functionality and which can also be easily called by less experienced developers and developers interested in the LarKC plug-in API only.

Taken together, these techniques for code development have ensured that it is possible for developers to work only LarKC related problems, being insulated from the more unusual aspects of Cyc code, while maintaining the possibility of fixing bugs and adding newly identified features from LarKC into Cycorp's Cyc code base, for purposes of exploitation and reuse.

This approach will also help developers even after all of the platform features have been fully defined and there is no longer any direct LarKC need to maintain the means access to additional Cyc code; the separation will allow them to work on core platform functionality and higher platform logic almost at the same time.

4.4. Plug-ins and use cases implementation

In general plug-in implementers from WP2 and WP4 have successfully implemented plug-ins using the design decisions that were based on the above requirements analysis. The full list of plug-ins is available at <http://wiki.larkc.eu/LarkcPlugins>. Furthermore, as mentioned in section 2.2, the Urban Computing use case (WP6) has also successfully implemented a complete pipeline, with its corresponding plug-ins. Lessons learned based on plug-ins (from WP2 and WP4) and Urban Computing implementers' feedback are described below:

- A need was expressed for how-to guidelines, common usage examples, templates, etc on both the plugin APIs and the Data Layer, confirming requirement PLAT_S_01 above.
- A widespread need was expressed for a richer mechanism to pass control and configuration parameters into plug-ins, confirming the need for a "context" datastructure.
- Similarly, the current API must be extended to allow passing back information on progress and resource consumption. An experiment is currently going which is abusing the Data Layer for this communication.
- The typology of the plug-ins was found helpful, but not completely clear-cut in all cases. This is a typical case of finding a balance between usability and reusability. Discussion on this is ongoing.
- Functionality for error-handling and throwing of exceptions is currently underdeveloped, and needs attention in the next version of the platform.
- The WP6 use case implementation confirmed the need to manage the execution of different plug-ins at the same time and the collection and merge of their results.
- There is also a need for support on concurrency, data persistence and data caching.

4.5. Code repository and Issue tracking

The LarKC Development Environment was created at the beginning of the project as a shared repository for the LarKC developers. As described in [12], this is implemented using GForge and is hosted by HLRS. It can be accessed through: <https://gforge.hlr.de/projects/larkc/>. Subversion (SVN) is used for the storing and versioning of source code, documentation, third party software and other file-based resources.

A common directory structure was initially defined for all LarKC components in order to promote standardisation and allow all partners to easily locate files and resources and thus encourage collaboration. The directory structure has evolved, according to the needs of the project developers, as the code implementation has progressed. A major change took place after the first LarKC WP5 Architecture Workshop (March 2009, Stuttgart), when a clearer split was defined between platform

¹ This is currently how the full Cyc system is implemented and deployed



and plug-ins functionality. This split had to be reflected in the code structure and therefore in the repository structure as well.

As developments were progressing, code files increasing and the LarKC developers' community growing, it became necessary the use of an issue tracking tool. The GForge issue tracking system has been adopted for management the request of new features and reported bugs of the prototype implementations.

The establishment of communication channels to support internal developers and external early adopters of the LarKC project is also needed, as the project internal WP5 list is not anymore suitable for this purpose. Different channels have been tested internally and it is being discussed which is the most appropriate way to support external developers. Currently the following tools are setup:

- LarKC Developers Forum and mailing list: for discussions about implementing new LarKC components (including plug-ins, pipelines/DECIDE plug-in, other possible components) or improving/modifying existing ones)
- LarKC Users Forum and mailing list: for discussions about using LarKC with implemented use cases (and corresponding implemented components), requesting new use cases to be implemented by LarKC developers, etc.
- LarKC Issue Tracker, which enables everybody with valid access rights (currently the LarKC developers) to raise issues (bugs, problems, requests for features, maintenance work, etc.) and keep track of their resolution status.

Discussions are being held about how to approach the upcoming increase of Early Adopters and how this may affect the current code repository and other supporting tools.

4.6. Licensing Issues

The licensing policies applied for the LarKC platform and for LarKC plug-in components developed within the project have been defined as part of WP9 tasks and are enumerated in deliverable D9.4.1 [13].

As development has progressed, the licensing goals of the project, which have sought to maximise both openness and breadth of contribution, have driven our selection of externally developed support code on which dependencies are possible. Licence monitoring is being scrupulously performed so that every component contains all necessary information to ensure that no dependency (either internal, to other LarKC component, or external to non-LarKC libraries) is created that may produce either an actual or conflict with the general LarKC licensing policy or its specifics.

4.7. MaRVIN and IBIS

We have been experimenting with building a distributed system for configurable RDF processing in the context of MaRVIN [11]. MaRVIN stands for "massive RDF versatile inference network", it allows logical inference over large amounts of RDF data using a versatile and distributed approach. See [11] for more information and results. The architecture and design of MaRVIN shares many features with the current LarKC architecture. In this section, we regard MaRVIN as one of the LarKC prototypes that explored the design space, focused specifically on the issue of distributed computing.

MaRVIN uses a decentralised peer-to-peer model to process massive amounts of RDF data. MaRVIN is designed to run on a local compute cluster or on a global grid. MaRVIN is completely configurable: all components conform to well-defined interfaces and each interface is implemented by several components. Through configuration files users can choose the specific components that should be instantiated for their run of the platform. MaRVIN has no central coordinating structure: all compute nodes are fully autonomous.

Similar to the main LarCK architecture, MaRVIN consists of a "platform" part that is responsible for data routing, inter-node communication and coordination, and a "plug-in" part that does the actual



data processing job. The "plug-in" part wraps existing reasoners (such as Sesame, OWLIM, or Jena) to compute the deductive closure of a given set of input data.

Grid environments often contain heterogeneous computing hardware, and since network connectivity is often an issue in these environments (for example, some grid resources are located behind firewalls, others cannot be reached over TCP). Therefore, MaRVIN uses the IBIS middleware [8] to abstract and overcome hardware heterogeneity and connectivity issues.

Within the first year of the project, we have build MaRVIN specifically to gain experience with the aspect of efficient models for distributed processing of RDF, while we aimed to minimise the integration effort needed to wrap existing reasoners. In the following, we summarise some of the lessons learned:

1. First, for a large-scale deployment in a wide-area setting, nodes should be autonomous and have symmetric functionality (i.e., nodes should not be special). The reason for this is that in a large-scale system, nodes should be expected to fail (nodes simply fail sometimes, due to hardware or software problems, especially if one is wrapping third-party code such as reasoners that cannot be controlled). If nodes are expected to fail their functionality should be generic so that remaining nodes can take over from failing nodes.
2. Secondly, communication should be asynchronous as much as possible, to reduce inter-node dependencies and bottlenecks in the system. If communication is synchronous, fast nodes must wait for slow nodes. This also means that each node should use multiple parallel communication ports to communicate with other nodes (otherwise, fast nodes still have to wait because slow nodes are blocking the communication ports of some node that both want to communicate with).
3. Thirdly, for scalability, the load (in terms of CPU, memory use, network bandwidth, and disk I/O) should be well-balanced across the nodes. In an unbalanced distribution some nodes become the bottleneck for scalability. Maintaining load-balance is not trivial though: in MaRVIN we first achieved load-balance through random exchanges (which decrease computation efficiency). We have then put considerable design effort into a more efficient strategy while maintaining load-balance.
4. Finally, as mentioned before, one should not underestimate the hardware and configuration heterogeneity in a wide-area network. With different network substrates, different protocols, and different configurations, many nodes cannot communicate to each other directly, and many communication channels require significant management effort to maintain. Therefore, using some network communication middleware that abstracts and resolves these issues is essential (for which we have used the IBIS middleware).

At the time of writing, an important emerging requirement is for the platform to support speculative knowledge manipulation (sometimes called "rumination"). The "MarVIN" experiments at VUA are an initial example of this. This manipulation is not driven by any particular query, but the likely needs of operations that LarKC "pipelines" will perform in the future, and represent an important class of optimisation (this optimisation is, for example, a reason why CycEur's Cyc inference engine supports large-scale forward inference including proof generation). If this requirement for speculative inference and transformation (at least) is adopted, it is likely to have significant consequences for the volume of data to be persisted, and therefore for the design evolution of the data-layer.

5. First draft of the LarKC Architecture

After a thorough analysis of the requirements described in chapter 3 and considering the lessons learned during the first project year, a first draft of the LarKC architecture has been produced.

Figure 4 shows a detailed view of the LarKC Platform architecture.

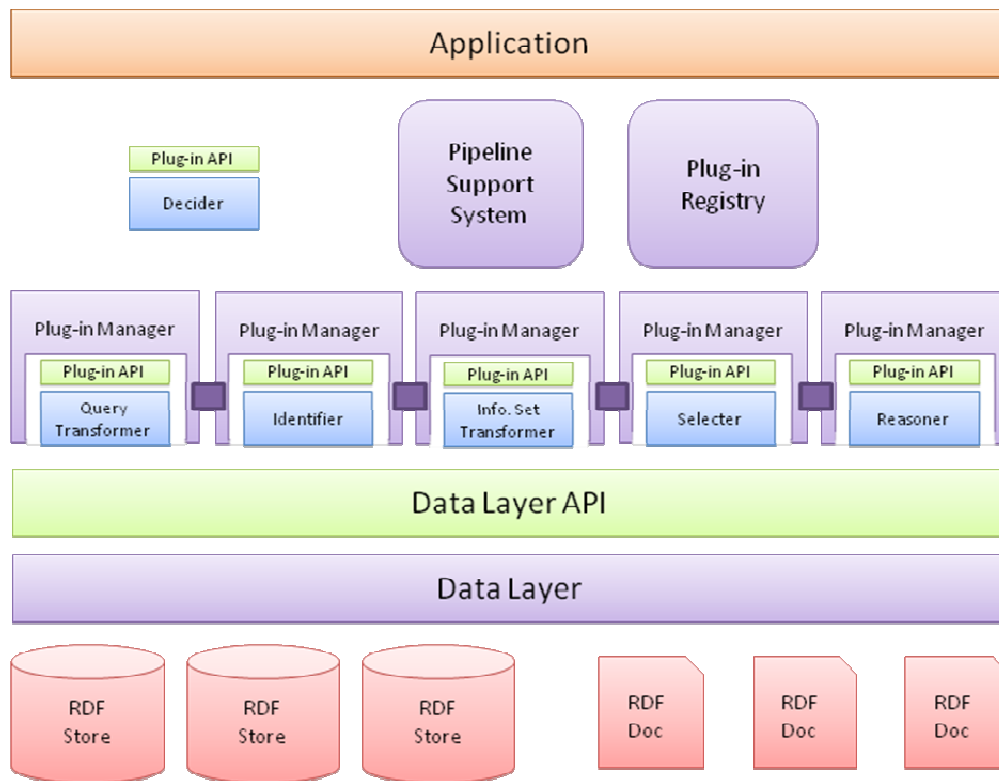


Figure 4: Platform Architecture – Main components

The LarKC platform has been designed in a way so that it is as lightweight as possible, but must provide all necessary features to support both users and plug-ins. For this purpose, the following components are distinguished as part of the LarKC platform:

- Plug-in API: it defines interfaces for required behaviour from plug-in and therefore provides support for interoperability between platform and plug-ins and between plug-ins.
- Data Layer API: the Data Layer provides support for data access and management via its API.
- Plug-in Registry: it contains all necessary features for plug-in registration and discovery
- Pipeline Support System: it provides support for plug-in instantiation, through the deployment of plug-in managers, and for monitoring and controlling plug-in execution at pipeline level.
- Plug-in Managers: provide support for monitoring and controlling plug-ins execution, at plug-in level. An independent instance of Plug-in Manager is deployed for each plug-in to be executed. This component includes the support for both local and remote deployment and management of plug-ins.
- Queues: provide support for deployment and management of the communication “pipes” between platform and plug-ins and between plug-ins.

Furthermore three different domains can be distinguished, which related software components may be deployed in the same location or remotely from each other:

- User Domain: user of the LarKC platform. According to the usage patterns, we may have three different kind of users: plug-in writers, pipeline constructors and end users requesting a query and expecting an answer. Each of them should be able to make use of LarKC either through a concrete application or a user interface.
- Platform domain: all platform related components are part of this domain.
- Plug-ins domain(s): all different kinds of plug-ins supported by LarKC belong to this domain.

6. Conclusions

After the first fourteen (14) months of the project a wide set of requirements has been identified and a first design of the overall architecture has been drafted and implemented. This has been an intensive



process combining rapid prototyping, requirements identification and architecture definition, in an iterative way.

Some of the identified requirements have been already fulfilled in the Early Prototype, which public release is being presented to the research community during the Early Adopters Workshop, on 1 June 2009, in Crete.

Some other requirements are going to be incorporated in future versions of the LarKC prototype, with the intention to have most of the requirements fulfilled in the last version of the prototype, towards the end of the project.

Next immediate step in the LarKC Collider Platform will be to continue developing the Architecture in order to design a complete and detailed view which fulfils all requirements. This process must include a careful traceability of the identified requirements in order to avoid losing any of them in the way. The architectural results will be incorporated in the different versions of the prototype.

7. References

- [1] <http://www.larkc.eu>
- [2] LarKC D1.2.1 Initial Operational Framework, http://www.larkc.eu/wp-content/uploads/2008/11/larkc_d121_initial-operational-framework_final.pdf
- [3] LarKC D1.3.1 Initial Plug-in Annotation Language, http://www.larkc.eu/wp-content/uploads/2008/01/larkc_d131_initial-plug-in-annotation-language.pdf
- [4] LarKC D5.1 Summary of parallelisation and control approaches and their exemplary application for selected algorithms or applications, http://www.larkc.eu/wp-content/uploads/2008/10/larkc_d51_summary-of-parallelization-and-control-approaches-and-their-exemplary-application-for-selected-algorithms-or-applications.pdf
- [5] DataSet definition: <http://www.w3.org/TR/rdf-sparql-query/#rdfDataset>
- [6] LarKC D1.1.3 Initial knowledge representation formalism, http://www.larkc.eu/wp-content/uploads/2009/01/larkc_d113-initial-knowledge-representation-formalism_m7.pdf
- [7] <http://www.ontotext.com/ordi/>
- [8] <http://www.openrdf.org/>
- [9] <http://www.cs.vu.nl/ibis/>
- [10] <http://sindice.com/>
- [11] <http://larkc.eu/marvin>
- [12] LarKC D5.6.1 LarKC Development Environment Available, <http://www.larkc.eu/wp-content/uploads/2008/07/larkc-d561-larkc-development-environment-available.pdf>
- [13] D9.4 1st draft of Exploitation and IPR Plan, http://www.larkc.eu/wp-content/uploads/2008/01/larkc_d94_1st-draft-of-exploutation-and-ipr-plan.pdf
- [14] <http://www.openmp.org/>
- [15] <http://www.mpi-forum.org/>
- [16] NetKernel Community, <http://www.1060.org/>
- [17] <http://boinc.berkeley.edu/>
- [18] LarKC D6.3 Urban Computing Environment Specification, due end May 2009
- [19] SATIN Programmer's Manual, <http://www.cs.vu.nl/ibis/downloads/satin/progman.pdf>
- [20] MAP-REDUCE tutorial, http://hadoop.apache.org/core/docs/current/mapred_tutorial.html
- [21] http://en.wikipedia.org/wiki/Amdahl's_law
- [22] <http://www.geant.net>