

Looking back on 10 years of the ATLAS Metadata Interface. Reflections on architecture, code design and development methods.

J Fulachier¹, O Aidel², S Albrand¹, F Lambert¹, on behalf of the ATLAS Collaboration.

¹Laboratoire de Physique Subatomique et Corpusculaire, Université Joseph Fourier Grenoble 1, CNRS/IN2P3, INPG, 53 avenue des Martyrs, Grenoble 38026, FRANCE

²Centre de Calcul IN2P3/CNRS Domaine scientifique de La Doua, 43 bd du 11 Novembre 1918, 69622 Villeurbanne cedex 69622, FRANCE

E-mail: jerome.fulachier @lpsc.in2p3.fr

Abstract. The “ATLAS Metadata Interface” framework (AMI) has been developed in the context of ATLAS, one of the largest scientific collaborations. AMI can be considered to be a mature application, since its basic architecture has been maintained for over 10 years.

In this paper we describe briefly the architecture and the main uses of the framework within the experiment (TagCollector for release management and Dataset Discovery). These two applications, which share almost 2000 registered users, are superficially quite different, however much of the code is shared and they have been developed and maintained over a decade almost completely by the same team of 3 people. We discuss how the architectural principles established at the beginning of the project have allowed us to continue both to integrate the new technologies and to respond to the new metadata use cases which inevitably appear over such a time period.

1. Introduction

This paper is about the development of the ATLAS Metadata Interface (AMI) framework [1] over the last decade. This framework is designed for generic cataloguing using relational databases. It is the basis of two tools which are part of the offline software of the ATLAS experiment at CERN [2]. AMI, the first and eponymous tool, is used principally to catalogue the datasets available for analysis. The other tool, named TagCollector, is a highly specialized application which is part of the management of the various releases of ATLAS software.

The functions provided by these tools will only be briefly described here, as each tool has already been the subject of previous publications [3][4][5]. Rather we will concentrate on the history and evolution of the project. The realization that the AMI framework is one of the longest surviving software projects within the collaboration was the catalyst for this approach. It is time to look back and record how the software has evolved over its lifetime, attempting to extract some of the principle ideas behind what we hope is a successful contribution to ATLAS.

Perhaps it should be mentioned that one of the specificities of the AMI Framework is that it was developed principally by a small team of non-physicist software engineers based at the LPSC Grenoble, France. In a recent and insightful article [6] the author states that HEP has “a complicated

and not very efficient relationship with Software Engineering”. Although we cannot deny some periods of frustration over the last ten years, we do believe that a good modular design, the basis of our framework’s structure, has served us well. Professional developers and physicists can work together! Not surprisingly however, the majority of our conclusions do echo those of the author of [6]:

- The importance of “mutual respect, dialogue and understanding”
- The importance of Agile [7] technologies
- The importance of a fast reaction to a bug or to a new use case.

In what follows we will begin by outlining the chronology of the project and describe the two software products. The main part of the paper will describe the underlying architectural choices we made at the beginning and how they have allowed us to adapt successfully to the inevitable changing and expanding demands of AMI and TagCollector users over ten very exciting years in the life of the ATLAS experiment. In the following section we will describe the hardware environment of the applications, and the working practices of the development team, in particular the tools used. Finally we will outline the major developments underway during the first long shutdown of the CERN Large Hadron Collider (LS1), and discuss the potential for the use of the framework in non-ATLAS contexts.

2. History

The first prototype of AMI was an electronic bookkeeping application for the ATLAS Liquid Argon detector component in 2001. Subsequently the work was adapted for the bookkeeping of the ATLAS data challenges. In parallel, the Grenoble team was working on TagCollector, a rather different application for release management. It was realised that since both applications used databases, both had web interfaces and they would be both used by ATLAS collaboration members, much of their underlying software could become common, and the AMI Framework, designed and developed between 2002 and 2004, was the result.

In 2006, following a review by the ATLAS collaboration, the AMI framework was chosen for physics metadata catalogues. This was the starting point for the expansion of the Dataset Discovery application.

During the same period TagCollector has also been considerably enhanced in order to deal with the increasing complexity of ATLAS releases.

3. Architecture and technology

The most important architectural and technological choices made for the AMI framework were guided by the context of very large and widely distributed scientific community over at least a decade.

We chose to implement a central web service used by lightweight clients. The enormous increase in the number of web applications in the 2000s comforted our choice and many of the technologies we decided to use, like Java/Apache Tomcat or XML/SOAP have become well supported standards and have facilitated the integration of AMI in GRID middleware.

AMI has a multi-tier client-server architecture in which presentation, application processing, and data management functions are logically separated. This model favors the creation of flexible and reusable applications because the modification or addition of a specific layer does not affect the rest of the application.

3.1. Use of the Open-Closed Principle

We have tried to adhere to the Open-Closed Principle (OCP) [8] as far as possible. Software entities should be open for extension, but closed for modification. To this end JAVA was chosen for our server side software. It is object oriented, portable and well adapted for both web applications and database connectivity. Encapsulation limits overall system complexity, and thus increases robustness, by allowing the developer to limit the inter-dependencies between software components.

To ensure OCP within our layered architecture we introduced dependency injections using some “inversion of control” design patterns, in particular the “bridge” or “plug-in” pattern [9] for

communication with external tools, notably with databases and version control engines. This pattern is a very good example of encapsulation; the plug-in architecture completely hides the internal methods and access is only allowed through the interface.

3.2. Database Schema Evolution

Schema evolution is inevitable during the application life cycle, therefore it must be considered when designing a database application. AMI compatible databases are constructed with a description which allows the code to discover the structure and to construct *ad hoc* SQL queries.

It has never been necessary to copy old AMI data to a new schema. Old data stays in the schema where it was written. The framework can construct threaded queries over both old and new metadata schema taking into account the evolution of ATLAS data

It is sufficient to add a description of a database schema to AMI to be able to browse the database tables in the AMI web interface. The schema description is internal for the AMI native catalogues, and external for other databases. This feature is exploited in our web interface by providing links to non-AMI applications such as the ATLAS production system database.

3.3. Other Characteristics of the AMI framework

In this section we describe the architectural decisions which have given the framework both strength and flexibility.

3.3.1. Database Connections. AMI uses indirection for database connections. The physical connection parameters are held in a central AMI database which we call the "router". Applications connect using only logical connection parameters. Thus database connections are completely transparent for users.

3.3.2. AMI user management. Users are managed centrally as shown in figure 1. Several authentication mechanisms are supported, including Virtual Organizations (VOMS) [10]. Some AMI roles are directly available to users with a VOMS role in ATLAS.

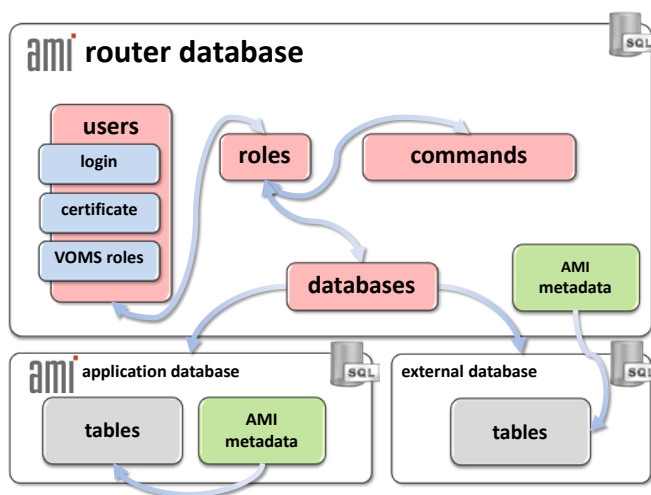


Figure 1. The users are mapped to a set of roles; each role is linked to a set of commands. The command execution may be limited to a sub-set of catalogues.

Users must possess a certificate known to VOMS, but as long as these credentials remain valid they may also authenticate with a username and password.

AMI also has a method for VOMS proxy forwarding, so that GRID middleware operations can be performed by AMI for users. This is typically the creation of dataset containers.

3.3.3. The command engine. The handling of commands is an extremely important part of the AMI framework. An AMI command can invoke a set of database operations on geographically distributed and different RDBMS (See figure 2).

Figure 3 shows the treatment of commands expressed using a generic Metadata Query Language [11] which is parsed and transformed to the appropriate SQL using the database descriptions described

in section 3.2. Atomic transactions are implemented using a hierarchical pool of individual database operations as shown in figure 4.

The native command output format is XML. It can be parsed to produce HTML, JSON or CSV/Text output. In order to speed the transformations we implemented an internal XSLT cached pool.

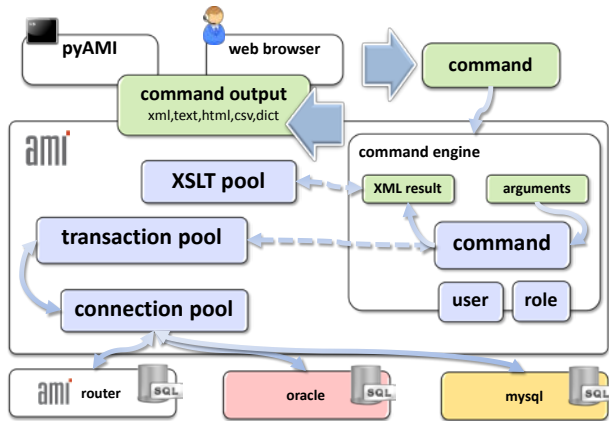


Figure 2. Gives an overview of the interaction of AMI layers. Client commands arrive either from a browser or the python client. After user authentication and authorization the command is executed by the command engine, possibly using several RDBMS through the transaction and the connections pools. Connections are made using the physical connection parameters in the router.

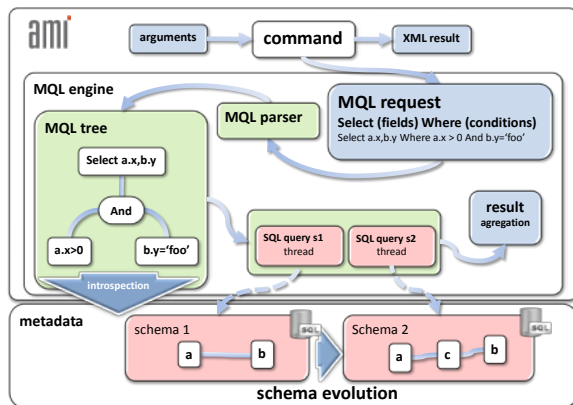


Figure 3. Outlines the transformation of a query in MQL to catalogue specific SQL queries, using database schema introspection, the results are aggregated to produce a simple row data output. This mechanism can be used as an abstraction layer in case of database schema evolution.

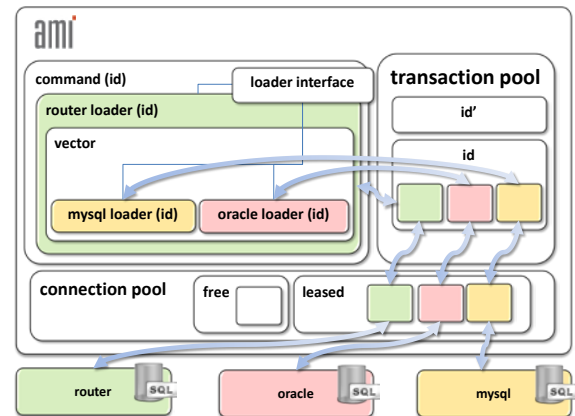


Figure 4. Transaction pool supporting a set of transactions. Each AMI transaction manages a pool of RDBMS connections and ensures atomicity at the command level. A command can contain calls to other commands belonging to the same transaction.

3.3.4. Web Applications and Web Service. AMI web applications are based on HTML/JavaScript produced by XSL transforms of the XML output of commands. We use Bootstrap, a standard CSS library [12] for formatting. AMI has its own AJAX command engine.

The AMI Web Service is standard SOAP [13]. The main client in HEP is pyAMI, a python client with a layer adapted for dataset discovery. The client is part of the ATLAS release and is supported by the AMI team. The pyAMI API allows ATLAS users to integrate calls to AMI in their own python scripts.

One of the advantages of the standard web service architecture is that most modern languages can produce clients based on a Web Service Description (WSDL). Users are not dependent on clients provided by the AMI team; clients have been produced for Ruby and GO for example.

4. Production environment

The AMI infrastructure (see figure 5) is designed to meet strong constraints of service availability and response time as befits an application with almost 2000 users situated all over the world.

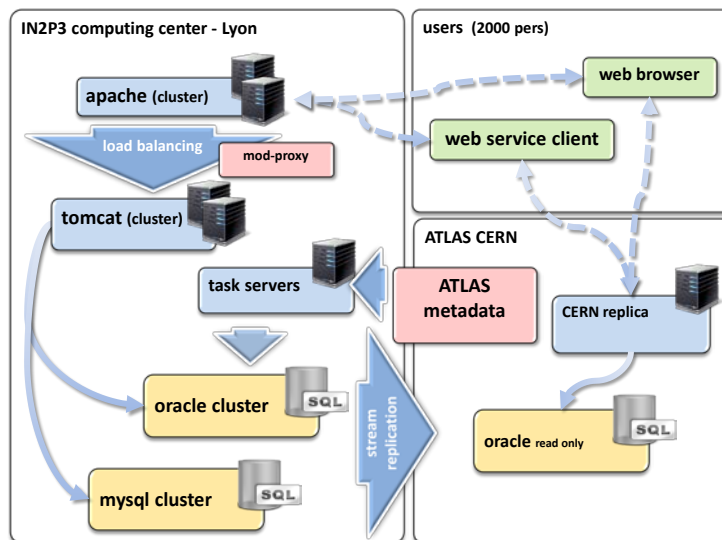


Figure 5. A summary of the deployment of AMI at the IN2P3 computing centre and at CERN. Users connect through Tomcat using either web browsers or the python client.

The task servers are instances of AMI running independently, used for filling the AMI databases from external sources, and for other recurring tasks.

The CERN replica is a read only instance, but it could become the production site in case of major problem at CCIN2P3.

The main site (<https://ami.in2p3.fr>), at the CCIN2P3 Lyon France, is hosted on an Apache cluster of 12 machines. It is configured to do load balancing using the mod-proxy module. User requests are dispatched over 4 tomcat nodes.

Additional redundancy is provided by a Tomcat server running on a virtual machine at CERN (<https://atlas-ami.cern.ch>), which accesses the read-only replica of the ORACLE database described in the section 4.1.

4.1. Databases

In this section we give some details of the AMI databases. Currently AMI uses both Oracle and MySQL hosted at the French Tier 1(CCIN2P3 Lyon). MySQL is used for TagCollector and some legacy dataset catalogues; the rest, and by far the larger part, uses Oracle. Each component has an architecture that designed to be scalable and robust.

The Oracle databases are hosted on an Oracle cluster of 4 nodes whereas the MySQL databases use a cluster with 2 nodes. All the database servers have redundant power supply and dual attachment on a SAN of 6 TB in RAID.

Over the past few years AMI usage has greatly increased, in 2010 only 40 GB were used against 400 GB in 2013. Data size and CPU usage often go together; in 2010 AMI only consumed around 3 percent of CPU in periods of high activity, but at present the CPU usage is on average 30%. In the near future, the CCIN2P3 will provide Oracle's standby database solution, used for disaster recovery and high availability. In case of crash or maintenance of the production database, all users could be automatically redirected to the standby database without any human intervention.

An additional level of security is provided in collaboration with the CERN IT. The mechanism used is based on the Oracle stream feature by capturing all updates run at CCIN2P3 and applying them to the CERN database. Oracle replication is unidirectional and writing on the CERN side is not authorized. A switch of master databases would take only a couple of hours.

5. Development methods and tools

At present the AMI project contains about 1200 Java classes, JavaScript, python and XSLT files. The team has 3 permanent developers and some collaborators who participate in the development of specific parts.

It was essential to establish a framework for efficient collaboration. To this end, we chose to use the following “standard” tools, which can often be linked to one another.

Subversion (SVN)	code repository & version control	https://subversion.apache.org
Eclipse	integrated development environment	http://www.eclipse.org
Firebug	web development tool (JavaScript debugger)	https://getfirebug.com
Twitter Bootstrap	CSS framework for the application web pages	http://twitterbootstrap.org
Sphinx	pyAMI documentation generation	http://sphinx-doc.org
Redmine	project management	http://www.redmine.org
Jenkins	continuous integration & deployment	http://jenkins-ci.org
Joomla	content management – used for the portal pages.	http://www.joomla.org

We focus here on two important points:

- Project management: keeping track of all issues (bugs and feature requests) and to have a global vision of the future evolution.
- Deployment procedure: linking the resolution of bugs or introduction of new features with code and release versions.

5.1. Project Management with Redmine

For project management we chose to use a forge, “Redmine”, a collaborative tool is installed at CCIN2P3 and shared by several laboratories of the IN2P3. This tool allows:

- Keeping a record the history of tasks such as bug fixes or feature requests.
- Tracing which source code was modified and why, by linking our SVN source code repository to a source code revision or to a specific ticket state change.
- Managing requests by setting priorities and categories and assigning them to a team member. Standard project management features such as the production of Gantt charts, are available.

5.2. Deployment with Jenkins

A rapid reaction to bugs is essential. This means that not only must the bug actually be fixed in the code, but also that the release and deployment strategies must be appropriate. We decided at the beginning of the project that our clients should be light, with a long client release cycle. The bulk of the work is executed on the servers. A classic “release schedule” is too unwieldy for this model so we decided to switch to a continuous release strategy.

We chose the tool “Jenkins” [14] a continuous integration server, to automatize application deployment. Jenkins is a system that allows us to build and deploy "on the fly" and “on demand” versions of our applications on our servers. It keeps the history the changes were made, and where and when the changes were deployed. The tool is accessible via a web interface and so allows developers to deploy from every place with an internet connection.

5.3. Programming Techniques

To extend development responsiveness and reduce maintenance cost we apply the Agile model [7] to our software development management. This model is well adapted to a project with a long life cycle and frequent feature requests. Some important points are:

- avoiding programming of features until they are actually needed
- programming in pairs
- including unit testing

6. Future and prospective

The LHC is currently stopped for a major upgrade, and is scheduled to restart at the beginning of 2015. During this time many operations will undergo major changes. The biggest changes that are needed in the AMI framework are in fact linked to changes elsewhere.

The ATLAS production system, one of our main sources of metadata, is being rewritten, and in consequence the data loading tasks must be adapted.

A more important set of changes required will come from the redefinition of the ATLAS dataset nomenclature, where AMI plays an important role as the repository of reference tables and software configurations.

We know also that the shutdown will generate new requests for very specific metadata functions from users, such as the Monte Carlo group.

The TagCollector web pages will be the subject of a review, leading to implementation of a complete new version. ATLAS release management policy has considerably evolved since the last version was implemented. The new TagCollector will take advantage of the dynamic pure JavaScript/Ajax interface developed for the Dataset Search in 2013.

We do not foresee changes to the lower level AMI framework beyond some optimization of code. However we have already started to implement some greatly improved performance monitoring tools which have already proved fruitful, for we have been able to understand some performance bottlenecks, and also study some user work flows.

Another interesting prospective is to better package the AMI framework so that it could be more easily shared with other experiments.

Acknowledgements

Over the years we have been helped and supported by many people in the ATLAS collaboration, and at the CCIN2P3 Lyon. We would like to thank in particular Philippe Cheynet, Benoît Delaunay, Pierre-Etienne Macchi, Yannick Perret and Jean-René Rouet of the CCIN2P3, Noel Dawe of Simon Fraser University and Asoka Da Silva of TRIUMF, Vancouver B.C. Canada.

References

- [1] <https://ami.in2p3.fr>
- [2] A Toroidal LHC Apparatus (ATLAS) <http://atlas.web.cern.ch>
The ATLAS Collaboration. The ATLAS experiment at the CERN Large Hadron Collider (2008) *Journal of Instrumentation*, **3(08)** S08003
- [3] Albrand S, Doherty T, Fulachier J and Lambert F (2008) J. Phys.Conf. Ser. 119 072003 (10pp)
- [4] Albrand S, Collot J, Fulachier J and Lambert F (2005) Proc CHEP (Interlaken, Switzerland, 27 Sep - 1 Oct 2004) pp.531
- [5] Emil Obreshkov et al. Organization and Management of ATLAS Software Releases; *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, Volume 584, Issue 1, 1 January 2008, Pages 244-251
- [6] Carminati F 2012 *J. Phys. Conf. Ser.* **396** 052018
- [7] Beck K et al. (2001) Manifesto for Agile Software Development. (Agile Alliance, <http://agilemanifesto.org>)
- [8] Meyer B 1998 *Object Oriented Software Construction* (Prentice Hall)
- [9] Gamma E, Helm R, Johnson R and Vlissides J 1995 *Design Patterns* (Massachusetts, Addison-Wesley)
- [10] Virtual Organization Membership Service (VOMS) <http://edg-wp2.web.cern.ch/edg-wp2/security/voms>
- [11] EGEE gLite Metadata Catalog User's Guide (2005) <http://www.egi.eu>
- [12] <http://twitterbootstrap.org>
- [13] Simple Object Access Protocol (SOAP) <http://www.w3.org/TR/soap>
- [14] Jenkins <http://jenkins-ci.org>