Draft version 1.1

ATLAS NOTE

ATLAS-SOFT-COM-2014-048

2014-03-13

# ATLAS Future Framework Requirements Group Report

John Baines, Tomasz Bold, Paolo Calafiura, Sami Kama, Charles Leggett, David Malon, Graeme A Stewart, Benjamin M Wynne



**Abstract**

The Future Frameworks Requirements Group was constituted in Summer 2013 to consider and summarise the framework requirements from trigger and offline for configuring, scheduling and monitoring the data processing software needed by the ATLAS experiment. The principal motivation for such a re-examination arises from the current and anticipated evolution of CPUs, where multiple cores, hyper-threading and wide vector registers require a shift to a *concurrent* programming model. Such a model requires extensive changes in the current Gaudi/Athena frameworks and offers the opportunity to consider how HLT and offline processing can be better accommodated within the ATLAS framework.

This note contains the report of the Future Frameworks Requirements Group.

# Contents

# 1 Introduction

The ATLAS Future Framework Requirements Group was constituted in Summer 2013 and was mandated to:

1. Summarise the requirements from both HLT and offline reconstruction for configuring, scheduling and monitoring algorithms, and other related functionality that is felt to be relevant. These requirements may be documented in old material that need checking for current relevance and completeness, or they may need to be reverse engineered using the skill and experience of the group.

2. Consider how these might be accommodated in a common framework that supports concurrency and helps to achieve high throughput on many-core computers, such as the GaudiHive prototype [1][2].

3. In particular, consider how to minimise the need for extensions or layers to the framework specific to one or other use cases, with the aim of making it straightforward to write algorithms to work well in both use cases.

4. Converge on the union of the HLT and reconstruction requirements for a future framework, and an analysis of the technical feasibility of satisfying them with a single common framework.

5. The study group is encouraged to think beyond current implementations, recognising that some decisions made a long time ago and in the context of the Gaudi framework may not be applicable in the future.

6. The study group is encouraged to consult experts in the trigger and offline software communities.

7. The study group should take about 2-3 months and provide reports related to interim milestones.

   (a) 1st month: initial view of requirements and other progress

   (b) 2nd month: iteration on requirements, report of early stages of analysis and implications on new framework and other progress

8. Final deliverable: report containing requirements, analysis and any recommendations for the design of the future framework.

In practice the group only properly started its work in March 2014 and quickly concluded that the original timescale was overly optimistic to produce a report that adequately covered all areas. However, by reporting in 2014 we allow adequate time for the collaboration to consider the next steps in framework design and implementation (timescales discussed in Section 4).

## 1.1 Methodology and Source Material

The group's methodology was to first have some general meetings to discuss scope and identify different topics that needed to be examined in detail. Then, in follow up meetings, the group either discussed particular topics utilising internal expertise or invited an external expert to introduce a topic. Follow up discussions by email were frequently held.

The group's twiki page holds a record of meetings:

http://cern.ch/go/7Cxj

The group's mailing list, `atlas-sw-ffreq@cern.ch`, was archived.

## 1.2 Concurrency and Hardware Evolution

As already noted, the major driver towards a new framework implementation is the advent of multi-core CPUs, where throughput can only be increased by parallel execution. Here we briefly review the drivers for this trend and the factors that constrain the throughput achievable in high energy physics computing applications.

The computing power of a CPU is proportional to its clock frequency and the number of components (transistors) it contains. More transistors allow more complex operations to be performed in a single clock cycle and higher clock frequency allows more operations to be performed in a given time. Historically, clock frequency and the number of transistors in CPUs increased roughly proportionally. Overall power consumption rose, but this was partially mitigated by the shrinking size of transistors on the die and the lowering of CPU voltages. In this era the throughput of HEP code rose naturally, benefitting from this steady increase in computing power.

However, around 2005, these increases in power consumption (with the associated costs of operations and cooling) could not be sustained. Clock frequencies started to plateau while transistor counts kept increasing as per Moore's law (Figure 1). These new transistors were used for adding new instructions, wide vector registers and multiple CPU cores into the same package (which provided hardware multi-threading, Figure 2). This approach continues to increase the theoretical computational capacity of a CPU, but exploiting this increase requires a parallel processing approach.

Concurrent execution — parallel processing — can take many different forms, but breaks down roughly along these lines:

- When a large input dataset can be divided into independent sections, it can be analysed in parallel by separate copies of a program. These processes can run in different cores of a CPU, but they can equally well be run on entirely separate computers. Thus, multi-process computation predates multi-core CPUs, and has been used in HEP for years.

- Multi-core CPUs allow a single program to perform multiple independent tasks at once, in separate 'threads'. This is conceptually similar to multi-process computation, but allows sharing of common resources, particularly memory, between the threads. Threads may be used to perform completely different operations, or to perform similar operations on different inputs.

- Single Instruction Multiple Data (SIMD) processing allows a specific operation to be performed on several inputs at once. A program must specifically structure the data it handles in order to exploit this feature successfully. Vector registers are loaded with 2, 4, 8 or 16 simultaneous inputs, and an operation is performed on the whole register.

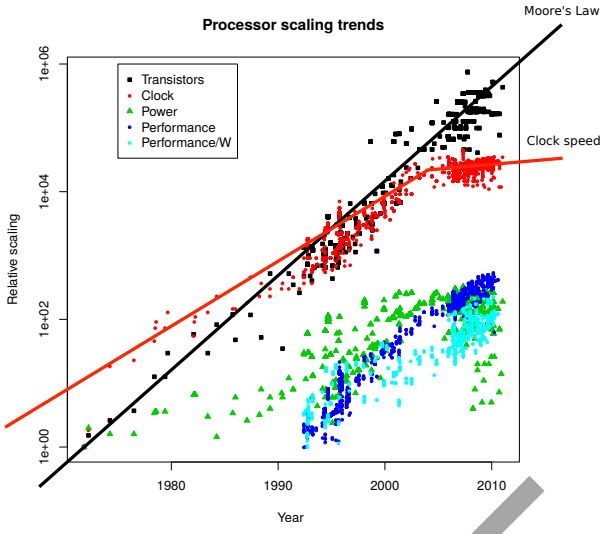


Figure 1: Historical CPU performance scaling [3]

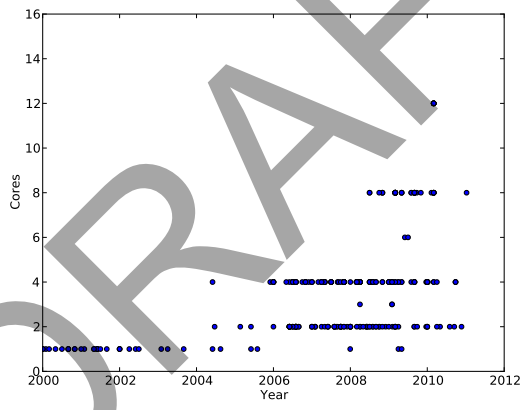


Figure 2: CPU hardware cores evolution [3]



- Co-processor cards tend to have very many processing cores, each individually unimpressive, but collectively powerful. These were originally designed to perform calculations on very large matrices (in computer graphics rendering), where usually each row, or even cell, in the input matrix can be treated independently. They are now used more generally for problems where the number of independent operations to perform is much larger than the number of available CPU cores.

The increase in parallel processing capacity has not been matched by proportional improvements in other system resources, such as RAM size or memory bandwidth. Over time, the power cost of memory and memory access has actually surpassed the power consumption of the CPU itself. Thus requiring large amounts of memory in low power, multi-core or many-core machine (to maintain a

ratio of RAM per core of 2-4GB[1]) is simply not cost or power effective. Consequently, the naive multi-processing model used in the past is becoming less feasible. This motivates a shift towards multi-threaded programming, where threads within a program can share memory, and towards more optimal memory layouts, where the bandwidth gap between RAM and CPU is mitigated by effective use of memory caches.

As it has become increasingly hard to engineer a single, practical die that provides excellent performance in all areas of computational space, computing architectures are also now becoming more diverse. x86_64, ARM64 and PowerPC are all present as general purpose CPUs and, while not directly connected with the framework, utilising diverse architectures is an important part of ATLAS's ability to exploit resources in the future, so this motivates ensuring that the framework (and more generally the software stack) does not impede our use of these resources.

These general platforms are increasingly backed up by specialist co-processors that themselves come in diverse forms: Xeon Phi, Nvidia GPGPUs, AMD GPUs and even FPGAs supporting OpenCL. These devices generally have wide, maskable, vector registers, a very large core count and total compute powers as high as multi-TFLOPs. Device memory per core is highly variable (100kB to a few GBs) and is often core local with very high performance. They can also be connected to the server with many different technologies. So, while it is not at all obvious that there will ever be a single, general, solution to utilise such devices, an updated ATLAS framework should certainly not hamper access to them and must consider their use in any future computing strategy.

Considering all the above, a new framework should help facilitate the shift from multi-process to multi-threaded processing for ATLAS, and, at the same time, should provide access to co-processor cards. Use of SIMD processing is somewhat outside the scope of the framework, however, consideration should be made for promoting the use of data structures that can easily be loaded into SIMD vector registers as the utilisation of optimised memory layouts is critical to the efficient use of modern CPUs and co-processors.



# 2 Current Framework

## 2.1 Offline Processing

### 2.1.1 The Gaudi Architecture

Athena is the ATLAS framework that implements and extends the Gaudi component architecture [4][5]. It is designed for ease of use by physicists, hiding implementation details behind abstract interfaces, yet flexible enough to allow the replacement of back end components, with minimal user impact, as new technologies emerge. It maintains a clear separation between *data* and *algorithms*, using object oriented design philosophies, and also enforces separation of transient and persistent data. Gaudi is implemented as a *state machine*, which is user extensible.

### 2.1.2 State Machine

During the course of a job, the framework takes its components through a series of *states*, as shown in Figure 4, which are `Offline`, `Configured`, `Initialized`, and `Running`, via a sequence of transitions. Components implement these transitions via specific methods, such as their constructor, `initialize()`, `start()`, `execute()`, `stop()`, and `finalize()`, which are called by the framework at the appropriate time. While the list of states is not extensible without modification of the framework,

[1] Here we also note in passing that the challenges of pileup in Run 3 and at HL-LHC only increase pressure on memory.

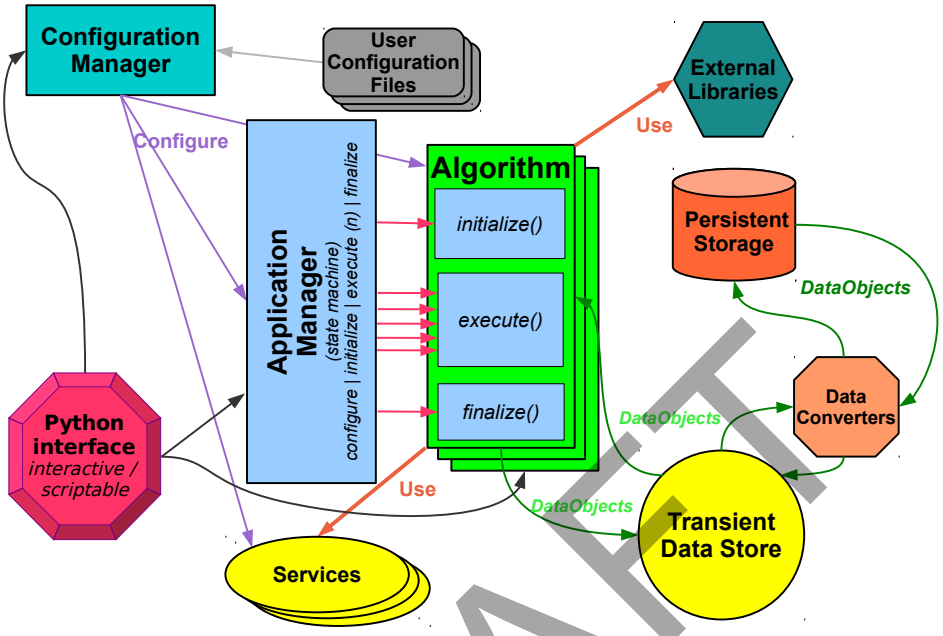


Figure 3: Gaudi Component Model

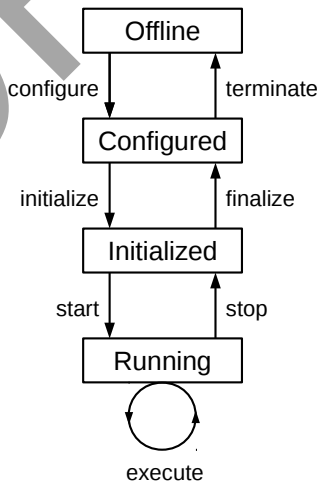


Figure 4: Gaudi State Machine

a similar effect can be achieved via the use of *incidents*, which are fired by the `EventLoopManager` at well defined times, such as `BeginEvent` and `EndEvent`. Not all states need be implemented by any component, in fact some components are forbidden to implement certain states, such as the `execute()` method for services.

### 2.1.3 Main Components

The main components of the Gaudi framework are Algorithms, Services, AlgTools and Converters. These components are accessed via abstract interfaces, to hide implementation details and allow transparent evolution. A plugin mechanism handles the loading of the various components and libraries, and a set of managers controls the creation, scheduling, and deletion of these components.

Data objects are the building blocks of the event. They can be event related items, such as tracks or hits; or specific to the detector, such as geometry objects. The type of object will determine its lifespan — event related data is cleared at the end of every event, whereas detector objects are much more static, and only change when conditions require. These objects also live in separate Data Stores to ease access and maintenance.

Algorithms are user-written elements that are responsible for manipulating data objects, or converting one type of data object into another. They are executed once per event, and implement a number of states.

Services are components that are setup and initialized once at the beginning of the job (or sometimes created on demand), and can be used by many other components. There is usually only a single instance of any one service instantiated at any time, and, once initialized, services have no state. Services can be retrieved by clients using service handles after initialisation. Accessing services is via abstract interfaces. A single concrete service can implement multiple abstract interfaces.

Tools are lightweight objects whose purpose is to help other components perform their work, usually by providing functions to process a particular type of data. They can either be *public*, and shared between clients, or *private*, where each client receives a new instance of the tool. Similar to services, tools are intended to be stateless after initialization and are usually accessed via abstract interfaces. Clients retrieve tools using tool handles (internally, the tool handle accesses the tool service, `ToolSvc`). In current ATLAS use we note that stateful tools are often used to communicate data between different framework components.

ATLAS has implemented various useful extensions to the Gaudi core algorithm, tool and service classes (`AthAlgorithm`, `AthAlgTool`, `AthService`, respectively). These extensions handle many boilerplate operations, such as access to event and detector stores as well as the message logger.

Converters are in charge of converting specific event or detector data into other representations, such as from persistent to transient, when reading data from offline storage, or the reverse when writing it out. Converters are specific to the data type that they are in charge of converting. They are automatically triggered by the framework when a new representation is requested.

Managers are specialized components that serve to orchestrate other framework elements. The *application manager* directs the operation of the job, loading the configuration, initializing the system, and deciding which algorithms to create, and when to call them. The *service manager* is in charge of creating and providing access to all services. The *algorithm manager* has a similar functionality for the algorithms. While there is no official tool manager, the `ToolSvc`, which for historical reasons is in fact a service, behaves in much the same fashion.

Auditors provide a mechanism to monitor the entry and exit points of certain methods, such as `initialize`, `start`, `execute` and `finalize`. While a number of these entry points are pre-defined in the `StandardEventType` structure, custom event types can also be defined by the user. Before and

after each of these methods is invoked by the framework, the `AuditorSvc` is passed a pointer to the component to be called, and the method. Individual auditors, which can perform disparate tasks such as measuring CPU time, monitoring memory consumption, or just printing the name of the component that is about to be executed, are then invoked, once again taking the pointer to the component and the method as parameters. The list of auditors to be executed is configurable at run time, and individual components can enable or disable their execution for specific methods.

### 2.1.4 Job Configuration

One of the key requirements for ATLAS software is the ability to assemble and configure an application without having to recompile any code. This is accomplished by setting run-time properties of Gaudi components, and their managers, via the `IProperty` interface. Properties are read in and set during the configuration stage of a Gaudi application. Initially properties were read parsing a set of ASCII job options files with an ad-hoc syntax. ATLAS Trigger replaced job options files with a set of database tables.

The configuration of ATLAS offline applications proved to be too complex to be captured by a set of property declarations. For example, the best default values for a component may depend on the type of job, detector geometry, or even input data. Also properties of different components may need to be kept in sync, for example by selecting a set of cuts to be shared across multiple tools.

The approach chosen by ATLAS offline was to replace the original ASCII job option files with a python job configuration layer based on auto-generated `Configurable` objects, that capture the default values, allowed range, and documentation of all `IProperty` components, and on `JobProperties` objects, that can be used to configure multiple components at the same time. Tool and service handles are also properties.

Python job configuration significantly improves Athena usability by detecting many configuration problems during the first seconds of an Athena job, rather than after several minutes when a misconfigured component is first used, and by greatly limiting the amount of cut-and-paste configuration which was needed with ASCII job options.

Unfortunately, using a powerful programming language like python for job configuration is not without its problems, particularly when the configuration of a job grows organically to tens of thousands of lines of python code. The lack of a real job configuration framework, and of adequate auditing/debugging tools, has made the configuration of reconstruction jobs in particular a dark art understood only by a handful of experts.

### 2.1.5 Data Access

One of the main design principles of Gaudi/Athena is the separation of data and algorithmic objects. The former are simple, stable, physics data objects like cells, tracks, or electrons. The latter are components like algorithm and algtools whose job is to simulate, reconstruct, or analyze data objects, hence being both producers and consumers of data objects.

An algorithm or an algtool interacts with data objects through an API called *StoreGate*[6], which can be seen as an in memory database of data objects keyed by type and name. Besides providing an API, storegate manages the lifecycle of data objects, from disk to memory and from memory to disk, relying on Gaudi persistence mechanism to manage the transient/persistent conversion of objects in a technology independent way. Storegate is composed of

1. `StoreGateSvc`, a Gaudi service which provides the associative array functionality.

2. DataLink and ElementLink, two handle template classes which are persistifiable references to data objects and their elements.

3. ProxyProviderSvc another service which supervises the just-in-time T/P conversion of data objects.

4. ClassIDSvc a registry of unique numerical identifiers for data object types.

5. AthenaOutputStreamer an algorithm and an array of algtools which steers the writing of data objects out to disk at the end of each event.

Some data objects are based on conditions data that are stored in the detector store. These objects have a certain interval of validity (IOV), that is identified by the meta-data associated with the conditions data. Often there is not a one-to-one correspondence between the conditions data and the value of the data object, but rather certain calculations must be performed to fill the data object. The IOVSvc is used to manage this process in a manner that is transparent to the user. During initialization, data objects are registered against the conditions data they depend on, as are callback functions that are used to recalculate more complex data objects. Complex relationships between the objects, functions and the conditions data they depend upon can be built, and stored by the IOVSvc as a directed acyclic graph. At event boundaries, (or in some instances only at run boundaries, or even job boundaries - the checking interval is set at run time via a jobOption), the IOVSvc determines the validity of all conditions objects that it manages, and resets any associated data objects that have gone out of scope. It will also trigger the execution of the registered callback functions when the conditions data changes. The next time a user accesses one of these data objects, it will either have already been updated by a callback function, or be automatically reloaded.

### 2.1.6 Data Model Foundation Classes

**Polymorphic Containers** ATLAS reconstruction relies heavily on the usage of polymorphic containers. Polymorphic containers allow algorithms to iterate on objects of disparate types (such as CaloCell, TrackParticle, or Electron) using a common interface (INavigableFourMomentum). ATLAS introduced the DataVector template, a memory-managing polymorphic container, that also supports the creation of user-defined views of its elements, as well as container-level “inheritance”, that allows the retrieval of, for example, a DataVector<LArCell> from StoreGateSvc as a DataVector<I4Momentum>.

**Support for Complex Data Models** ATLAS has the capacity to employ transient data models of considerable complexity, models that reflect the formidable expressive power of C++. Supporting classes provide machinery and templates for streaming the states of such transient objects into state representations more directly amenable to persistence (and more directly suitable for object state transmission across networks or among processes). Additional infrastructure classes integrated with the framework’s conversion services handle object persistification and associated concerns. Importantly, this infrastructure also provides a natural locus for support of (arguably inevitable) schema evolution in both transient and persistent data models.

Data model infrastructure and supporting classes provide a means to create externalizable references to specific events and to use them as input to processing, identification of a primary event "entry point," a locus for recording the constituent content objects associated with a given event along with a means to navigate to that content, and a record of the provenance of a given event (e.g., the ESD or the

raw event from which the analysis content of an event was derived), and the means to navigate thereto. The ATLAS DataHeader and its supporting classes, and persistent object reference technology adapted from the LCG POOL project, provide this and related functionality.

The ATLAS data model provides a standardized means of event identification and event type discrimination, implemented via an EventInfo class that also hosts the principal, generally immutable quantities by which both simulated and real events may be efficiently selected or filtered. The EventInfo class has undergone some evolution in xAOD, but continues to serve the same purposes.

**Auxiliary Data** In C++, as it is an object-oriented language, it is natural to access the elements of a container as objects, and therefore to layout containers in memory as Array-of-Structures, or, in other words, object-wise. With the introduction of deep vector pipelines the wisdom of this layout started to be challenged. Laying down data as Structure-of-Arrays, or member-wise, allows GPU compilers to generate SIMD instructions (also known as PTX instructions) that accessed data stored contiguously in memory. Having all data members laying down contiguously in memory also allows compilers targeting current x86 processors to vectorize loops operating on these data members.

At the same time, as any ROOT user knows well, many analysis applications only access a fraction of a container (or n-tuple) data. The ability to read containers member-wise (branch-by-branch) from disk can speed-up an analysis by orders of magnitude.

Both these use cases are satisfied in ATLAS xAOD system in which data members of the elements of a `DataVector` are saved into separate "auxiliary store" containers. `AuxStoreInternal` objects have the member-wise memory layout that benefits vectorization, and I/O read speed.

Clients of `DataVector` will not be directly exposed to these `AuxStoreInternal` objects, and can use the traditional object-wise access pattern with negligible loss of performance.

### 2.1.7 Scheduling

**Event Loop** The EventLoopManager is the heart of the framework, and directs the execution of the various framework components during the course of a job. First it initializes basic services and the configured algorithms, then `starts` these components. Next it calls `execute` on all configured algorithms as many times as there are events requested by the job configuration, or until no more events are present in the input file. It monitors the execution of these algorithms, and will react appropriately if any fail, such as by terminating execution of the current event and skipping to the next one, or by halting the job entirely, and attempting to exit gracefully. Once all events have been processed, it will stop all components, then finalize them, leaving them in the `Offline` state, and closing all output streams.

**Incidents** The Incident Service is used by components to trigger asynchronous events in other clients, following the *observer pattern*. Components which are interested in a particular incident subscribe to it by name via the `IncidentSvc`. When that incident is fired, which is performed by calling the `fireIncident` interface of the `IncidentSvc`, all clients which have subscribed to that incident are called in sequence. In order to subscribe to an incident, clients must inherit from the `IIncidentListener` base class, and implement the `handle` method, which takes an `Incident` object as a parameter, and is called by the service when the incident is fired. The client must also tell the `IncidentSvc` of the type of `Incident` that they wish be informed of via the `addListener` interface. A client can subscribe to as many incidents as is desired, but then must test the value of the Incident

parameter within the `handle` method to implement the appropriate behaviour. Some examples of incidents are `BeginRun`, `BeginEvent`, `EndEvent`, `AbortEvent`, `BeginInputFile`, `EndTagFile`, and `CheckIOV`.

## 2.2 High Level Trigger Processing

### 2.2.1 Key concepts in HLT events processing

The design and construction of the HLT framework used in Runs 1 and 2 is the result of a decade of R&D followed by a review [7][8][9] and the final implementation, from 2005-2007. The design was motivated by the online requirements and, in particular, the limitations of bandwidth and CPU resources. Driving concepts behind the design are incremental reconstruction and selection steps, that provide early-rejection, and reconstruction inside geometrical Regions of Interest (RoI) that correspond to part (or the whole) of the detector. These key concepts limit the reconstruction performed to the minimum needed to arrive at the trigger decision. This is especially important as ~99% of L1 triggers are rejected by the HLT. Rejection of events based on partial reconstruction, sufficient to disprove all physics signature hypotheses, is the main factor in saving HLT resources. Another key concept is the independence of trigger chains, which means that one trigger does not influence another. This provides the operational flexibility to add, remove or prescale triggers and aids analysis by facilitating the evaluation of trigger efficiencies (as the product of the prescales and the efficiencies of the L2 and EF chains and the L1 triggers seeding them). Also key is the ability to import offline tools into the online environment.

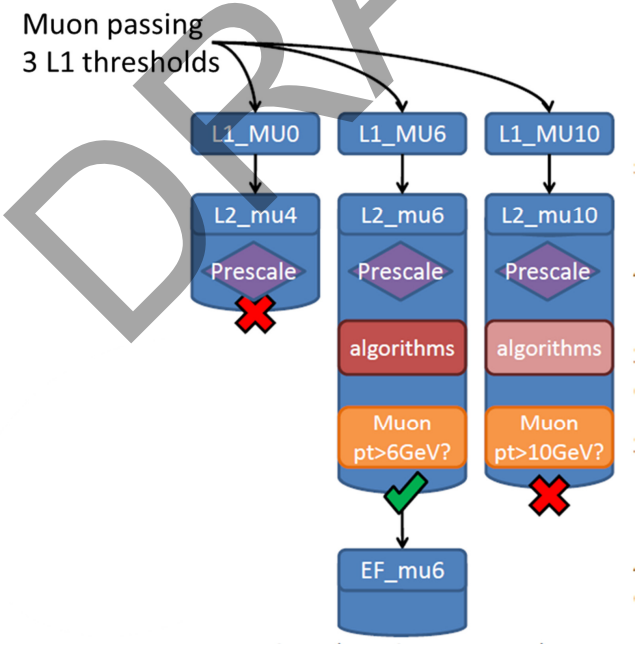


Figure 5: An illustration of the processing of three trigger elements (TE) created from a single muon passing three L1 thresholds. In this example, only the HLT chain *L2_mu6* is fully processed.

These requirements necessitated a number of additions to the offline framework: a trigger-specific scheduler, the HLT steering; wrapper-algorithms (`HLTAlgo`), to allow online and offline tools to be used in the online context; and Trigger Elements (TE) and Trigger Navigation, to provide context during the online selection and in offline analysis.

A consequence of chain independence is that separate chains are defined for each trigger threshold. This means that several TE can be created from a single RoI. This is illustrated in Figure 5 which shows that three TE are created from a single muon passing three different L1 thresholds. The diagram also illustrates the concept of partial reconstruction. In this example, only the HLT chain `L2_mu6` is fully processed. The other chains are deactivated, the `L2_mu4` chain by a pre-scale and the `L2_mu10` chain by a failed trigger hypothesis. In order to avoid duplicate reconstruction in the two chains executed on the same RoI, the HLT employs caching of reconstructed features. The caching is implemented in the HLT Algorithm base class and ensures that the same reconstruction step is not run twice on the same input.

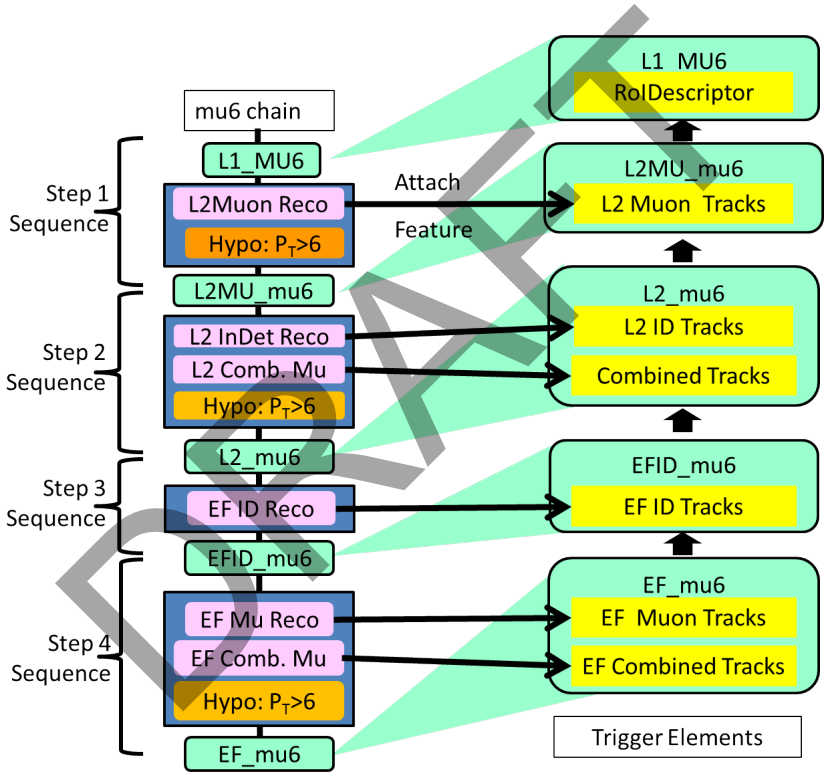


Figure 6: A diagram representing a muon trigger chain, showing the trigger elements (green), algorithms (feature extraction: pink, hypothesis: orange) and reconstructed trigger objects (yellow).

### 2.2.2 Trigger Configuration

The trigger is configured by a menu that defines a set of trigger chains (often referred to simply as triggers) that start from a L1 trigger and specify a sequence of reconstruction and selection steps that result in the reconstructed physics objects (or signatures) required by the chain. A trigger chain might require a single signature (e.g., a single muon passing a $p_T > 6$GeV threshold, `mu6`) or a signature

and multiplicity (e.g., `2mu6`) or a logical combination of signatures (e.g., a trigger requiring a tau lepton and muon, `tau60+mu20`). A diagrammatic representation of a muon trigger chain is shown in Figure 6. Chains are composed of a number of sequences. The input to and output from the sequences are TEs. The TE are objects providing the RoI context that are passed from the HLT Steering to the HLT algorithms. Reconstructed trigger objects (or features) are attached to the TE. A sequence is an irreducible component of the trigger processing that is uniquely specified in terms of the algorithms it runs and the input and output TEs. A sequence typically consists of one or more *feature extraction* (FEX) algorithms performing reconstruction and a selection step performed by a *hypothesis* (HYPO) algorithm.

The size of the HLT menu defines the scale of the task performed by the framework. The menu consists of primary triggers that are the principal physics triggers, supporting triggers (e.g. orthogonal triggers for efficiency measurements and pre-scaled lower threshold versions of the primary triggers), monitoring and calibration triggers. The latter are used to collect specialized datasets consisting of partial event data from a sub-set of detectors. A typical Run 1 menu consists of about one thousand chains and a factor of two to three more algorithm sequences (although for the majority events only a fraction of these are run). The trigger configuration also defines the HLT prescales. These provide an additional way to optimize the system for high rejection. The L2 and EF prescales are applied prior to the L2 or EF processing.

### 2.2.3 HLT Steering

The HLT steering is the algorithm responsible for the execution of the menu. The L1 result defines RoIs that form a set of TEs that seed the HLT reconstruction. Execution proceeds starting from the chains activated by the L1 seeds. At each step, all the sequences are executed for all of their specified input TEs (i.e., all the RoIs of a given type). After each step, the number of active TEs in a chain is tested against a multiplicity requirement to determine whether processing of that chain should continue or not. This process is repeated for all steps and all active chains.

### 2.2.4 HLT Algorithms

HLT algorithms are derived from Athena algorithms where the `execute()` method is replaced by an `hltExecute` method that is invoked with additional arguments informing the algorithm about the execution context (i.e., the RoI). There are three sub-classes of FEX algorithms allowing an execution on a single TE (`FexAlgo`), a combination of TE (`ComboAlgo`) and an algorithm allowing arbitrary use of all input TEs (`AllTEAlgo`). The later two algorithm classes allow information from different RoIs to be combined. The B($\mu\mu$) b-physics trigger is an example of a `ComboAlgo` that is executed on all combinations of two different muon RoIs. An example of an `AllTEAlgo` algorithm is the algorithm that combines information from all muon RoI in order to correct missing $E_T$. Another `AllTEAlgo` algorithm is used to re-run jet-finding across all Jet RoI. The HLT algorithms implement caching of reconstructed features. If the same algorithm is invoked more than once on the same input the features are retrieved from the cache.

The HLT uses a number of algorithms from the offline, e.g., tracking at the EF, and integrates them so that they can be run on RoIs. The integration involves custom fitting for each signature and somewhat *ad hoc* solutions have been adopted. In the new framework, it should be possible to run offline algorithms unchanged in the online environment.

### 2.2.5 Trigger Navigation

HLT algorithms communicate by passing Event Data Model (EDM) objects through the auxiliary structure of trigger navigation (`AlgTool`). This structure implements a directed acyclic graph in which the nodes are TEs and the edges are the seeding relations of the HLT sequences which materialize in a given event. The trigger navigation is bootstrapped by the information from L1 and further extended during execution of HLT algorithms. EDM objects produced and requested by HLT algorithms are attached and retrieved from the navigation using the TE (token) passed at execution. The memory management of these EDM objects is left to the `StoreGate` service. The navigation structure is also queried by the HLT steering in order to provide the input TEs for sequences and in order to count multiplicities. For accepted events the trigger navigation is serialized and saved to allow this information to be queried during analyses. Additionally, the trigger navigation is responsible for manipulation of the trigger EDM:

1. merging collections from multiple RoIs and performing the associated book-keeping.
2. filling the event store with empty EDM collections when the HLT algorithms that would create them have not run in a specific event.
3. interfacing to the infrastructure that serializes the HLT objects that form the HLT result that is stored as part of the RAW event (bytestream).

### 2.2.6 Optimization of Event Processing

In order to provide early rejection, reconstruction proceeds in an incremental way. After each increment of reconstruction, a selection is performed. If the evaluation based on the reconstructed objects within the chain does not pass the selection, there is no further processing of that chain. If there are no active chains in the event, the event is rejected. When processing chains with signature multiplicities greater than one or with a logical combinations of signatures, the reconstruction may be abandoned in the early steps as soon as the multiplicity or combined requirements are not met. This is particularly important as full processing of single low-threshold trigger items could otherwise be very costly.

Incremental reconstruction is also important to reduce data-request rates since the data needed for steps later in a chain are only requested if the earlier selection steps are passed. For example in the muon trigger chain, shown in Figure 6, muon detector information is requested and reconstructed in the first step and a selection is made based on this information. Inner detector information is only requested, reconstructed and combined with muon detector information if the earlier selection is passed.

A further consideration is the order of execution of the sequences, i.e., step-wise or chain-wise processing. Step-wise processing means that step-n is executed for all chains before step-(n+1) is run. In the alternative, chain-wise processing, a chain is processed from start to finish (stopping if a selection step fails), before moving on to the next chain. In both modes the data requested and the final level of reconstruction are identical, but step-wise processing allows an optimisation of data-access by grouping data-requests from different chains and across RoI. This is possible because many chains are arranged similarly, in that they perform similar reconstruction in the same chain steps. For example electron and tau chains both perform calorimeter reconstruction in the first step and inner detector reconstruction in a later step. The execution of a given step is preceded by a pre-request for the data fragments, stored in the Readout Buffers (ROB), that are needed in that step. Given this information, the framework ensures that the data requests to the Readout Systems (ROS) are performed only once for all potentially needed ROBs and, therefore, the request rate is reduced (one ROS serves several ROBs). The use

of step-wise processing gave an important reduction in data-request rates in Run 1 and Run 2. But it imposes a synchronisation across chains that would constrain the parallel scheduling of algorithms in the new framework. It would be possible to preserve the advantage of grouping data-requests in a chain-wise processing model if data for all steps and all RoI were pre-requested prior to the start of event-processing. But this would significantly increase the rate of data requests for some systems, particularly the inner detector. While upgrades to the ROS and network may make data-request rate limits less of a constraint in Run-3, the system should also scale to Run-4 where L1 rates will be significantly higher. It is, therefore, important that the new framework has the flexibility to support both step-wise and chain-wise processing modes to enable an overall optimisation of the system to be made based on operational experience.

Other optimizations have also been implemented, i.e., the order in which the chains are processed is chosen so that those requesting the biggest chunks of data are executed first (e.g., jets before electrons).

### 2.2.7 Integration with Monte Carlo production

The HLT (and L1) trigger run as part of MC production. Since the HLT has been developed within the Athena framework, it is relatively easy to construct a combined job that runs both trigger and offline reconstruction. In this case, the HLT reconstruction (`HLTSteering`) is one of the algorithms of the offline algorithm sequence. This mode of operation was used until recently when, due to virtual memory constraints and the necessity to run trigger and reconstruction using different software releases, MC jobs were split into two with the trigger running before the offline reconstruction. Although separate offline and trigger jobs will continue to be needed in specific cases, the ability to run trigger and offline reconstruction together should be preserved within the new framework. As described above, there are additional requirements associated to the scheduling of HLT algorithms (such as RoI context) on top of the requirements for running offline algorithms. A single component of the new framework (the *scheduler*, §3.1.2) should take into account all these requirements in order to support the scheduling of both offline and HLT algorithms.

### 2.2.8 Additional Trigger requirements

The trigger imposes requirements of the framework that are in addition to those of the offline uses cases. The key additional requirements and constraints are summarized below:

1. Limits on the average per-event execution time determined by the L1 rate and HLT farm size: Run 1: L2 40 ms/event, EF 4 s/event; Run 2: 240 ms/event.

2. Rejection: Full HLT reconstruction is only performed for approximately 1 in 100 L1-accepted events.

3. Early termination of chain processing: Reconstruction terminates as soon as the reconstructed objects fail a selection step.

4. Reconstruction inside an RoI: The trigger performs partial reconstruction of the event inside geometrical regions. This greatly reduces the per-event execution times and hence required farm CPU resources since the RoI correspond to typically only about 10% of the full event. It also reduces the amount of the data which needs to be requested from the ROSes in order to make the decision. However, by Run 3 ROS upgrades could permit event building at the full L1 rate.

5. Forced accept: Possibility for a chain to force acceptance the event.

6. Error handling: Possibility for an algorithm to trigger routing of the event to the debug stream (algorithm returns an error code containing the desired *action* and *reason*).

7. Streaming: The trigger routes events to different file-streams dependent on the trigger result. The framework should support different streaming policies such as inclusive and exclusive streams and specialised streams containing partially built events.

8. Processing of many runs by a single job: Typically several runs will be processed between the configure and unconfigure state transitions. However, it is also possible for a new job to be started part way through a run, e.g., in the case that errors cause a HLT node to be rebooted.

9. Conditions: Most conditions changes only permitted at `prepareForRun`: during Run 1 and Run 2, most conditions updates only occur at the `prepareForRun` state transition. Only menu pre-scale changes and very limited small conditions changes (a limited subset of conditions folders that were of small size, e.g., beamspot position update) were permitted during a run and only at a luminosity block boundary. Configuration from a database, rather than from python. Configuration identified by three integer keys: Menu key, L1 prescale key, HLT prescale key.

# 3 Requirements for Event Processing

*In the following description of framework elements, we consider that any framework element or feature that is not explicitly stated to be* optional *is* mandatory. *These elements and features will provide the necessary coherent architecture to support the algorithmic code and tools needed by ATLAS. This set of features should eliminate the need for algorithmic code to duplicate or circumvent the framework.*

*Within the text we occasionally make some recommendations about patterns for utilising the framework to its best advantage.*

## 3.1 Required Framework Elements

The model for event processing in HEP is mature and has existed for many years across different experiments. Concurrency does not alter the model for how events themselves are processed, which is illustrated graphically in Figure 7 for the offline case.

After some setup phase (*initialization*, including data-dependent initialization that requires access to the event stream), events are processed through a series of *algorithms*, which produce new derived data products. These algorithms can make use of general pieces of code, encapsulated as *tools*, if they are private to the algorithm, or as *services*, if they can provide data to all algorithms and events. After event processing has completed (and selected events and data products have been serialized), the job performs some *finalization* actions.

Use cases can exist for some non-event based communications between framework elements, which are handled using *schedulable incidents* (§3.1.9). However, as discussed in §3.3, any use of incidents should be minimized.

For use cases *online*, additional features are required, as outlined in §2.2.8.

### 3.1.1 Whiteboard

The whiteboard (or *event store*) is a service and is the main mechanism for algorithms and tools to communicate as *Event Data Model* (EDM) objects are exchanged through the whiteboard. The navigation

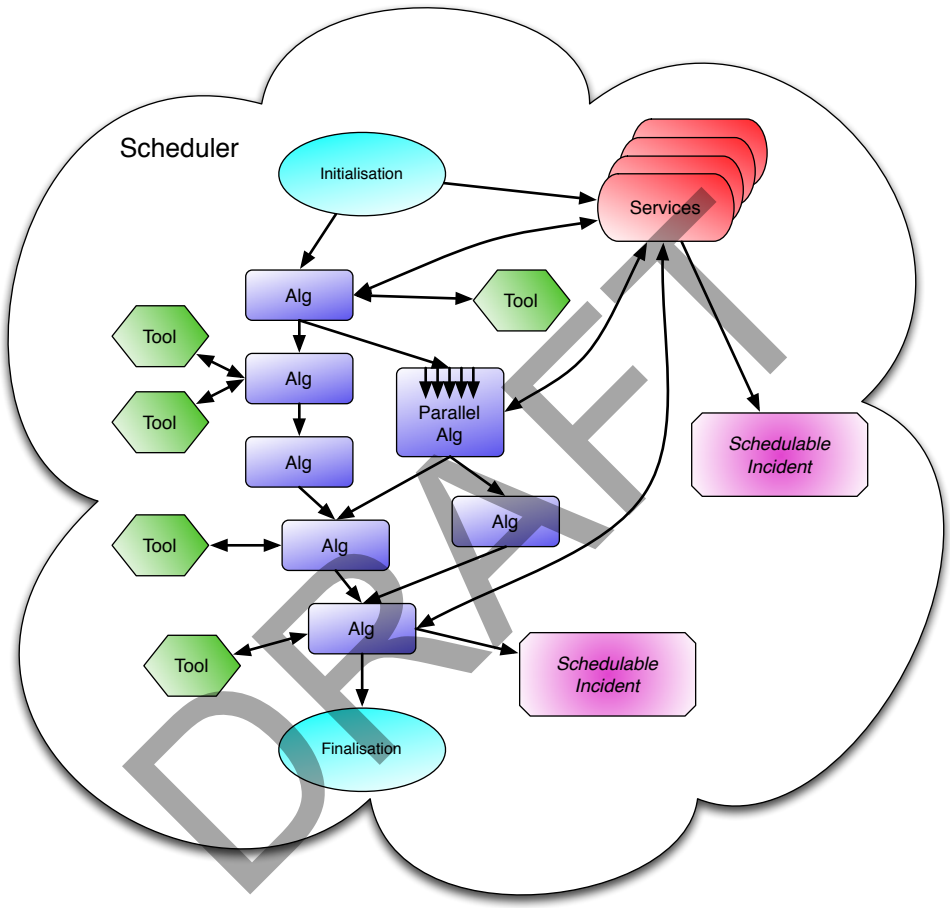


Figure 7: Schematic of offline framework elements and event processing workflow, showing algorithm and in-algorithm parallelism (for clarity, event parallelism is not shown here). Note that although flow is illustrated as one algorithm 'flowing' into another, this is really achieved through data objects that are stored in the whiteboard service. Execution of algorithms and incidents is under the control of the scheduler.

structure plays a similar role for the HLT, but with an additional RoI context. Harmonization between these two components is an essential part of the new framework.

1. The whiteboard is used to store data objects and exchange them between components.
2. Algorithms may record objects in the whiteboard as well as read existing objects. Algorithms may modify objects that they have created during their execution or that are being prepared as part of an algorithm sequence (see §3.1.4).
3. Modifying objects in the whiteboard is ideally limited to augmentation: adding new information to an existing object. Overwriting existing data or deleting parts of an object's data is permitted.
4. Deletion of an object in the whiteboard is permitted, which allows algorithms and tools to use the whiteboard as a 'scratch' space to communicate data between themselves.
   (a) Care should be taken not to delete objects used by downstream algorithms.
   (b) The whiteboard may optionally be configured to delete objects when the last algorithm which uses them has run. This will reduce the event store memory footprint during event processing.
   (c) Contrary to this, the whiteboard may be configured to disallow the deletion of objects.
5. Data objects will be marked as immutable after all declared algorithmic writers have been executed.
6. All access to EDM objects should be through the whiteboard API.
7. A view object, with the same interface as the whiteboard is available. It contains a selection of the data objects in whiteboard, usually in a particular geometrical region. Each algorithm or tool should be able to operate on a view in the same way as the whiteboard. Views can be created by any component and can be interconnected by N-to-N relations.
8. Data for multiple events and different views can exist in the primary whiteboard at the same time, allowing concurrent processing of these events or views.
9. After an event has been processed the framework will clear the whiteboard of that event's data.

Additional requirements on the efficiency of the whiteboard can be made:

1. The whiteboard must be accessible by concurrent threads: data race conditions expressed through data dependencies will be prevented by the scheduler, and concurrent read operations must be supported.
2. Internal data storage in the whiteboard should be optimized to allow efficient access and processing on modern hardware, taking into account the memory hierarchy. This means that storage of vectors or array of contiguous *plain old data* objects must be supported. Note that it is the responsibility of algorithm writers to design efficient objects, bearing in mind how the data will be used. (This development should have a very high priority in the upgrade as adaptations to new EDM are potentially among the most disruptive. The Run2 xAOD is a good starting point.)

3. Internal data storage should also allow efficient transport of data blocks to/from accelerator devices (§3.4.3) with minimum overheads for conversion between layouts. (This item is *optional*, in so far as actual use of accelerators remains undecided.)

4. The implementation of event views must also add minimum overheads to processing, when compared with direct access to the same underlying memory locations.

All of the above points should be addressed by early demonstrators of the whiteboard and of the event views. In particular, efficiencies should be compared between direct memory access to an optimal layout and access via the whiteboard, with and without, event views. Overheads of a few percent will be considered acceptable.

### 3.1.2 Scheduler

The scheduler controls the use of available resources that need to be marshalled to complete a job. Specifically, it will control the start of each event though the processing chain and determines the order of algorithm execution during event processing. The scheduler also controls execution of other framework elements that may arise in a less predictable fashion: schedulable incidents (§3.1.9), such as service triggered conditions data preparation (§3.4.2). The scheduler's general goal is to maximize throughput, while respecting the configured constraints (e.g., the maximum number of events in flight, or certain algorithms or resources that are not thread safe). However, the scheduler is a pluggable component of the framework, so specialist schedulers are possible.

1. The scheduler is a regular component of the framework, consisting of exchangeable elements configured in a similar way to any other component.

2. The scheduler keeps track of data objects in the whiteboard, configured algorithms in the job and defined sequences of algorithms.

3. The scheduler will mark algorithms as executable once all their specified resource requirements are satisfied. These will almost always include *data dependencies*, but additional *control flow* conditions may be specified as well as any special requirements (e.g., use of a thread-unsafe external library or a special piece of hardware). Control flow is important for implementing early rejection in the trigger (e.g., object multiplicity is one example of a possible extra condition). Control flow conditions can also be signalled by algorithms in the case of errors.

4. The scheduler will execute ready algorithms or sequences by submitting them to an execution engine, which may have its own task queue.

5. The scheduler will only execute an algorithm once for a particular data input. An algorithm can be scheduled multiple times per event if it operates on different inputs each time (i.e., data from different RoIs).

6. It will be possible to schedule parallel execution of algorithms within the same, or different, events. This includes:

   (a) Different algorithms running concurrently.

   (b) Copies of a particular algorithm concurrently analysing data from different events.

   (c) Copies of a particular algorithm concurrently analysing different data inputs from within the same event.

In all cases this must be moderated by the thread-safety and clonability properties that each algorithm declares.

7. If an algorithm modifies a data object in the whiteboard it will be scheduled before any algorithms that only read the object (unless that algorithm is bound into a sequence, §3.1.4).

8. It should be possible to disable concurrency in the scheduler either by configuration or by providing a replacement trivial scheduler.

9. The state of the scheduler should be recordable. It should be possible to replay the sequence of algorithms back into the scheduler to help with debugging.

10. The scheduler's handling of errors propagated from algorithms should be configurable, e.g., error conditions could abort processing of a view, prompt an event abort, cause the event to be written to a special error stream or cause a job abort.

11. It must be possible to run trigger and offline algorithms in the same job.

12. Schedulable incidents will be processed and handled in a timely manner and will not cause overall throughput to drop unnecessarily (if processing if blocked because of incidents requiring slow external services to respond this is considered unavoidable from the scheduler's point of view).

### 3.1.3 Algorithms

The algorithm is the basic schedulable unit within the framework event loop.

1. Algorithms manipulate data objects in the whiteboard. Data objects to be accessed and created will be published by the algorithm when it is initialized. Data dependencies of all tools used by an algorithm will be inherited by the algorithm.

2. The recommended way of adding objects to the whiteboard is to prepare the object privately, then to add it to the whiteboard when the object is ready. If needed, modifications to compound objects can be allowed once they are in the whiteboard, e.g., objects can be added to a collection even after registration. Multiple write operations (create/modify) of the same pieces of data in the whiteboard by the same algorithm are permitted, but discouraged.

3. All levels of thread safety and/or clonability are supported for algorithms:

   (a) Ideally algorithms should be thread safe, allowing multiple instances to be used freely by the scheduler.

   (b) If algorithms require specific configuration, differently configured instances should be independent of one another. This allows the scheduler to clone multiple instances for parallel running.

   (c) Algorithms can also be thread-unsafe and uncloneable. Though generally undesirable, this will allow for a gradual migration of algorithmic code to a new framework so must be supported.

   (d) In order for an algorithm to run concurrently, all the components that it calls, such as tools, or external libraries, must also be concurrent capable. This parallelizability of framework components must be communicable to other components.

In all the above cases, algorithms must declare their level of thread safety and clonability to the framework.

4. Algorithms may utilize parallelism internally, but should always do so using concurrency facilities provided or brokered by the framework (to avoid over-commitment of resources). An algorithm should declare that it uses internal parallelism at initialization.

5. Since there can be a number of views in an event, algorithms are not be limited to a single execution per event. Algorithms should be capable of working on fragments or subsets of event data. The same fragment should be processed by the tools and services it utilizes to perform the task.

6. It must be possible to schedule multiple instances of an algorithm with different configurations, then have the scheduler treat these as independent algorithms. (E.g., to allow optimised configurations of the same algorithm for different trigger sequences.)

7. Configurable properties of an algorithm may not change after the configuration step; nor may an algorithm change the properties of tools it owns or services it uses. Algorithms should have well designed and meaningful configurable properties.

8. Algorithms should be equipped with appropriate monitoring, providing information in greater detail than that published to the whiteboard and sufficient to validate the correct operation of the algorithms and debug problems.

9. Algorithms should be able to return a detailed error status to the scheduler, with enough information to allow appropriate error handling to be performed by the framework.

### 3.1.4 Sequences

A sequence is a list of algorithms that must be executed in order.

1. Sequences are to be used when several algorithms modify the same data object and the order of execution plays an essential role.

2. A single algorithm can be included in multiple sequences.

3. A sequence should be treated as a single task by the scheduler.

Sequences may also incorporate algorithms that only read a particular data object (but may need to do so before a later algorithm in the sequence modifies them).

### 3.1.5 Tools

Tools should be viewed as configurable parts of an algorithm allowing the implementation of data manipulation in a generic way (such that it is useful for multiple algorithms and merits encapsulation). The execution of tools is not dictated by the scheduler.

1. An algorithm may delegate the execution of a specific task to any number of tools. Instances of tools may not be shared between algorithms (i.e., public tools are prohibited).

2. Whether tools are executed, and in what order, is determined by the parent algorithm. Tools called by the parent algorithm may themselves call other tools.

694 3. When an algorithm is cloned, the tools it owns will be cloned as well.

695 4. In order to maximize the ability to run concurrently, Tools should be stateless and thread safe
696 after their configuration. If a tool is not stateless and thread safe, then any other component
697 which uses it, such as an algorithm or another tool, cannot be run utilising a level of concurrency
698 that the tool does not support. This information must be communicable to the tool's callers.

699 5. The EDM data exchange between tools and components other than its parent algorithm (or in-
700 termediate tools) happens through the whiteboard. A tool's caller is allowed to pass information
701 directly to a tool as method arguments using the tool's interface.

702 6. Tool dependencies on whiteboard data are declared and will be propagated upwards to the parent
703 algorithm. This propagation will work also in the case of nested tools.

704 7. Unlike a service (§3.1.6), a tool is only used by its parent algorithm or its parent tool.

705 8. Like algorithms, tools may also utilise parallelism services managed by the framework.

706 ### 3.1.6 Services

707 Services control access to resources that are necessarily shared between multiple events and will be
708 accessed from different running threads. Consequently, each service may only have a single instance
709 within the framework. Any service must be accessible by concurrent threads, with any conflicts man-
710 aged internally. With the notable exception of the whiteboard, services may not be used to pass data
711 between algorithms – any algorithm's interaction with a service should be independent of previous
712 interactions. Example service tasks include:

713 1. Event related storage services (§3.1.1)

714 2. Disk I/O

715 3. Offloading tasks to co-processors

716 4. Database and conditions access (§3.4.2)

717 5. Giving access to other large static data structures in memory, e.g., geometry or magnetic field
718 map.

719 When an algorithm or tool interacts with a service a suitable *event context* is passed, so that the
720 service can return the correct objects or values (e.g., the correct conditions data for that event, §3.4.2).

721 ### 3.1.7 Auditors

722 Auditors are framework components that monitor resource consumption of algorithms. They gather
723 and collate operational performance data, such as execution time, memory consumption and name.
724 They are required as part of the overall monitoring infrastructure provided by the framework.

### 3.1.8 Converters

Converters provide technology specific implementations of the I/O layer (see §3.4.1), allowing clear separation between the persistent and transient representations of the data. By implementing abstract interfaces, and using the Conversion Service, clients are able to stream object states and read and write data without explicit knowledge of the data format on disk or over the network. Converters are replaceable components, such that as technologies evolve, they can be replaced with minimal impact on user code.

Converters should be configurable at runtime, like any other framework component.

### 3.1.9 Schedulable Incidents

A schedulable incident can be triggered by other framework elements and causes a sequence to be added to the scheduler's task list. Thus, operations that must happen upon reaching a condition not known to the primary data/control flow (e.g., opening or closing a file) will be executed in a timely fashion, but under the control of the scheduler.

As noted in §3.3, use of incidents is strongly discouraged in favour of data/control flow scheduling as the framework element that triggered the incident may have to block until the incident handling task has been completed.

## 3.2 Overall Framework Features

There are some requirements concerning all elements of the framework. They are listed here.

1. The state machine of the framework should be mapable onto the online (TDAQ) state model.
2. Development of trigger specific algorithms should happen within the offline framework. Only a small number of exchangeable components should be needed to make the system suitable for running online.
3. Any dependencies between components for configuration, initialization, finalization or termination need to be clearly expressible, so that any constraints may be respected. This allows for possible parallelization of these tasks. Components should not rely on an ordering which is not derivable from a data or interface dependency.

## 3.3 Framework Rationalisation

Compared with the current implementation of ATLAS software in the current version of Gaudi/Athena, the following specific changes to the framework should be adopted:

- Public tools should be dropped from the framework and replaced with:
  - Private tools (or now just *tools*) where inter-algorithm communication is not necessary.
  - Services for the use cases where data is provided to multiple algorithms.
  - Communication via the whiteboard where data objects are prepared via a sequence of algorithms (specifically this pattern replaces the current ATLAS pattern of using a public tool to pass data between a sequence of algorithms).
- Sub-algorithms should not be supported. All use cases can be covered with algorithms and sequences.

- Incident handling becomes a process handled by the scheduler, but is strongly disfavoured when data/control flow can achieve the same result.

## 3.4 Additional Details

### 3.4.1 Input/Output Layer

Input and output pose fundamental challenges to the efficient exploitation of emerging computing platforms. Input and output are points of serialization, and I/O bandwidth has not scaled with processing power or core count. Applications whose computational elements may be scalable to very large numbers of CPUs will lose their scalability if they are in fact I/O bound. Even when this does not happen, a framework that achieves high throughput by creating a substantial post-processing burden (e.g., in a later, possibly complicated, merge step) has achieved that throughput only nominally.

1. The I/O layer should support variable numbers of readers and writers, both to provide a means to match I/O to processing capacity and to allow adaptation to a range of deployment environments.

2. The framework should be configurable to support I/O-intensive as well as CPU-intensive processing, without the I/O layer itself being the bottleneck.

3. While the architecture of I/O components may be complex, it should be factorized from the architecture of the scheduler: for example, readers and writers (event selectors and output streams) should be schedulable in essentially the same way as any algorithm.

4. The framework should be agnostic to the nature of its data sources and sinks, i.e., to whether data come from local or remote storage media or from specialized readout devices, and to whether they are written to storage or to local or remote processes. I/O layer components will deal with the necessary specializations and should be developed as required (thus specific components will be *optional*).

5. The I/O layer (together with the whiteboard) should isolate the algorithms, tools, services accessing data from the persistency technology used behind the scenes by using converters (§3.1.8). Current data formats, including bytestream, xAOD, and the POOL formats, need to be supported. Any explicit dependency on a particular persistency technology should be avoided as much as reasonably possible.

6. Input and output infrastructure must be capable of respecting semantic constraints on data organization, such as not interleaving events from different runs or run segments (luminosity blocks).

7. The framework needs to provide sufficient bookkeeping to ensure that all events in semantically meaningful units have been processed, and may be required to provide more detailed bookkeeping in jobs that filter events. The I/O layer should facilitate such accounting, and should provide a means to associate metadata with event samples.

8. The I/O layer should exploit similarities in HLT and offline data access where possible. There are parallels between data requests to readout systems and I/O requests to storage, as well as parallels between selective RoI retrieval and selective (partial event) retrieval from disk.

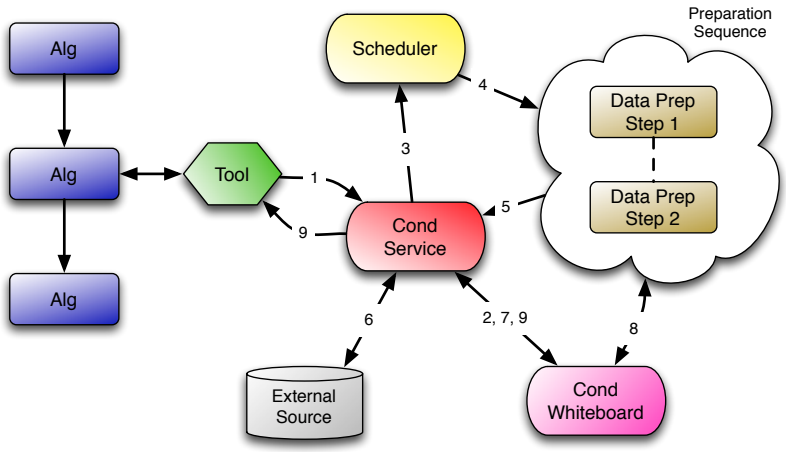


Figure 8: Schematic of time varying data retrieval: a tool asks for a piece of processed conditions data from the conditions service (1); the service checks the data is not found in the conditions whiteboard (2); the service then schedules a data preparation sequence (3), consisting of two steps; these steps are run by the scheduler (4), requesting the requisite raw data objects from the service itself (5); these objects are recovered from an external source and placed in the conditions whiteboard (6, 7); the data preparation components then run to prepare the calibrated data, utilising the conditions whiteboard in the usual way (8); the calibrated data is returned to the calling tool (9).

### 3.4.2 Time Varying Data

Data for processing events varies naturally though time. First, event data itself is time dependent, with each event occurring at a different time. This implies that associated metadata, describing the state of the detector, or other relevant quantities, will also change for each event. This data itself can come from various locations: local disk, a database connection or another network source. In addition, the data needed by the event loop may require some derivation or calculation from the raw data supplied to the running job; such calculations may themselves require additional pieces of time varying data.

Thus, time varying data should be supplied by a *service*, which is able to take event context into account. The time varying data itself may be stored in a *whiteboard*, separate from the event store. The service should be able to some trigger data preparation component, which can be used to ready the data as needed and these components can have dependencies in the usual way. Running this data preparation step will be delegated to the scheduler, which will handle this task in the same way as any schedulable incident, §3.1.9. This scheme is illustrated in Figure 8.

The service associated with the time varying data should manage the lifetime of the data in its whiteboard. Data may be retired after some time after it is last used or to ensure that the space occupied does not exceed some threshold.

### 3.4.3 Accelerator Devices

Co-processors, as noted earlier in §1.2, are increasingly an important part of delivering high performance computing as a part of exascale challenges. The diversity of devices leads to complications in coding paradigms, programming language selection, hardware aptitude to tasks and efficiency tuning; each of them typically requires a different approach. Even if ATLAS does not require the existence of co-processors for its processing, due to the shared nature of compute farms, and the existence of

opportunistic resources, it is likely that co-processor devices will be available for framework to use. Therefore the new framework should be able to make use of co-processors if they are present. Thus framework must be able to:

1. Function without the existence of co-processor.
2. Have a hook facility that will enable interaction with co-processors that are available at runtime.

Actual use of specific co-processor hardware is somewhat outside of the scope of this document, however we list here the following desirable, but *optional*, features should co-processors become a significant part of ATLAS computing:

1. Handle different types of hardware at runtime such as Xeon Phi, GPGPU, FPGAs or other new co-processors that might be introduced in the future.
2. Make use of multiple devices at the same time, such as embedded accelerators and add-on cards, taking their capabilities and processing powers in to account.
3. Allow event bunching to improve the efficient use of hardware, since offloaded tasks from a single event or algorithm might not have enough work for a co-processor to offset data transport costs.
4. Take event properties into account to do offloading and reduce overheads. For example, it may decide to offload events with high number of hits (tracks) to co-processor while process others in CPUs.
5. Support shared or partial use of the co-processors when the co-processor is shared between multiple processes or hosts as it is possible to share one co-processor among many hosts for efficiency and cost reasons.
6. Use co-processors in the most optimal way (hybrid, accelerator or as independent processor modes). Depending on the properties of hardware and availability of the algorithms, it may choose to use the co-processor for doing sub-algorithm processing, such as loops; sub-event processing, such as a set algorithms; or full event processing with one or many events.
7. Support libraries and languages that are not natively used in framework itself as long as they are ABI compatible. For example, the framework should be able to use algorithms written and compiled with CUDA or any other compiler or language.

### 3.4.4 Configuration

Configuration of a complex software framework is itself a complex matter. Historical experience shows that schemes that are too simple (e.g., ASCII or plain config files) rapidly fail to scale to the appropriate level. However, schemes that are too flexible, allowing any parameter to be changed at any time during the configuration process, can become impenetrable and entangled. This situation is particularly harmful to ATLAS as it erects a significant barrier for new users and developers of the framework. Given the turnover of ATLAS members, a comprehensible configuration system is a very important part of making the ATLAS framework more accessible.

Thus a new configuration scheme should:

1. Use a uniform scheme for each part of the job configuration (tools, services, data handles, etc.) as far as possible.
2. Be flexible enough to compactly enable dependent configuration of multiple framework components.
3. Implement a hierarchical scheme where general settings cascade in a well defined way to specific options, enhancing the comprehensibility of the configuration.
4. Allow for the visualization of a job configuration (*optional* item).
5. Ensure that components are configured in a well defined and reproducible way and signal a warning (or error) if a configured component's setup changes unexpectedly.
6. Be able to serialise a configuration, into a language neutral format.
7. Allow reloading of a previously generated configuration from multiple sources (file, database, network source, etc.).
8. Allow a job to be configured from a de-serialised configuration, with then a few further changes made at runtime (e.g., to change an input file).

We would recommend the continued use of python as the most generic configuration layer, but with the adoption of more structured components to achieve the above requirements.

## 3.5 EventService

The ATLAS event service changes the granularity of jobs from files to events. Events, or event ranges, are sent to Athena running in a client mode, somewhat like the configuration of the HLT. Input data is then read event by event and outputs are shipped off the processing node in a timely fashion for downstream merging. It offers advantages when running on opportunistic resources, such as some clouds or on HPCs and can increase flexibility when defining the number of events to be processed in a job. It also allows for a 'mixed mode' deployment of Athena on many-core architectures, where multi-process and multi-threading are used at the same time.

The principal points for a future framework, when considering the deployment of Athena in an event service mode, are:

1. Starting Athena in a 'client/server' mode should be simple, with the client scheduler processing events as they arrive and not terminating the job until a specific 'stop' signal is received from the server.
2. Event reading should be handled more efficiently by the new I/O layer, which can then be agnostic to the backend delivering event data.
3. Similarly, consolidation of outputs at the server can be managed better by a single I/O writing service that reads events over the network and manages merging on the fly as the outputs are serialised. This server may be a different process from the server reading input event data from storage.
4. In all cases metadata for the output files needs to be handled correctly, taking into account that events may be far more unevenly processed than in the case that parallel processing occurs on a single machine.

## 3.6 Code Evolution

When considering the requirements that ATLAS has for a new framework, it is necessary to bear in mind the very considerable amount of legacy code that ATLAS has developed before and during LHC data taking. While some code will naturally be deprecated and new code will replace it, we still anticipate a considerable amount of code will need to be migrated to the new framework.

The difficulty, or ease, with which such a migration can be performed will have a very significant impact on the effort ATLAS requires to move a new framework into production and, thus, on the overall success or failure of such an endeavor. Therefore this aspect of the framework, *ease of migration*, becomes itself an important requirement.

We therefore identify the following as being important requirements on how the framework interacts with ATLAS code that exists today:

- The framework must present interfaces that match those of the current Gaudi/Athena framework.
- Where needed, interfaces that offer better integration with the new framework (e.g., a parallel algorithm class) should be introduced without removing existing functionality.
- Existing algorithms and tools should need only be modified to remove behaviour prohibited by the new framework. The scheduler should control parallelism and concurrency such that these components will then function correctly, even when their implementations cannot be run in parallel.
- The framework and interfaces must fulfil both trigger and offline use-cases. Some changes will be required align trigger and offline interfaces allowing offline components to be used directly in the trigger.

In so far as the current usage of Gaudi/Athena is not compatible with the envisaged evolution, interface migration should be undertaken even in the current code. A good example of this is the ongoing migration of access to StoreGate to using data handles and the migration of tool and algorithm base classes to `AthAlgTool` and `AthAlgorithm`. Even in the parallel framework, eventual deprecation of older interfaces should be considered in order to prevent code bloat and maintain a healthy code base.

If these points are adhered to, then a gradual and incremental evolution to the concurrent framework, with the minimum possible code evolution, will be made easier.



# 4 Timescales

The timescale of future framework development is primarily driven by the time at which ATLAS needs to have the new framework in production, with migration of the algorithmic code completed and validated.

From the trigger side, this date is set for LHC Run 3, which is currently scheduled for 2020 [10]. As it is expected that the migration of algorithmic code is a very substantial undertaking, this would have to begin in 2018. Thus the new framework must be delivered and production ready by the end of 2017.

This would then imply the following rough outline for framework development:

- **2015 Q1-2** Decision on core technological solution and framework design.
- **2015 Q3-4** Initial framework prototyping, giving the ability to run basic tests by end of year.

- **2016** Continued development of framework, working in tandem with the evolution of a limited set of subsystem algorithms.
- **2017** Refinement of framework and bug fixes, gradual opening to more use cases and further development as necessary.

In the group's opinion the framework development should endeavour to work with real ATLAS algorithmic code at a very early stage, so as to ensure that development is driven by the requirements found from real event processing. It should also be the case that algorithmic code should avoid making unnecessary requirements on the framework and so close cooperation with framework developers will help to optimise use of the framework's existing features to satisfy event processing requirements.

# 5 Conclusions

## 5.1 Recommendations

The current evolution of hardware is forcing software to become more concurrent. For ATLAS to continue to make efficient use of computing resources that are available, it is therefore necessary for our software to evolve towards concurrency as well. Up to now, ATLAS event processing has been naturally parallelized at the event-loop level. A piecemeal evolution of parts of the algorithmic code, without proper framework support, would not be an effective way to proceed. Even when algorithmic code can utilise fine-grained loop-level concurrency this will not open up enough parallelism to properly exploit multi-core devices, due to Amdahl's Law (however, as we have stressed this parallelism is important to also exploit, but will be greatly aided by proper framework support). Therefore, we conclude that ATLAS should evolve its framework towards multi-threaded and multi-process execution of events and of algorithms within events. As this process will involve substantial rewriting of framework elements, we also highly recommend direct support for ATLAS HLT use cases in the framework. We believe that the technical challenges in doing this, while considerable, do not prevent ATLAS from taking this opportunity, which will benefit the long term health of the software by providing better core support and widening the developer base. Aspects of HLT framework support may also prove useful for future offline developments (e.g., event views).

The group has outlined the key requirements for this framework evolution in this document. In general, we recommend retaining the core design principles behind the current framework, but extending these to better incorporate requirements from the HLT, and better accommodate concurrent execution. We should adopt as many practical simplifications as possible, which both helps implement concurrent execution and achieve code maintainability. Certain highly desirable features, such as thread-safety in algorithms, are considered optional. This is mainly to ease the transition to a multi-threaded framework. However, time critical algorithms and tools must evolve quickly to a clonable or thread-safe state to reap the benefits of concurrency.

Evolution of the framework for the start of Run 3 does not give a lot of time for the design and development phases, considering the large quantity of existing code that must then be migrated or re-implemented. Therefore the next steps in the future framework process should be decided on quickly.

## 5.2 Observations

We make some final observations that may help to guide further progress:

- The Gaudi framework has not proved to be a limiting factor for ATLAS in Run 1, nor in preparations for Run 2. Therefore, we believe that evolving the current Gaudi/Athena framework towards concurrency is a good choice (especially considering the matters of code migration, §3.6). An evolution of Gaudi also offers continued collaboration with LHCb and CERN SFT, which is a substantial benefit.

- The amount of work needed to implement a new framework should not be underestimated. It will take a considerable number of skilled developers to evolve the framework, even given the progress made in the GaudiHive demonstrator.

- Most of the difficulties in transitioning to efficient concurrent processing come from the current algorithmic code and the data structures that are used. Thus, a substantial effort will be required for algorithmic changes and implementing new, improved, design patterns. Old code should be aggressively deprecated and backward compatibility should not limit future evolution.

- To achieve substantial in-algorithm parallelism, we note that in current ATLAS reconstruction (ttbar, release 20.1.2) more than 75% of the event loop CPU time is accounted for in the top 12 algorithms (of a total of 189). Thus parallelising these algorithms can potentially reduce the needed number of events in flight by a factor of 4, with the same number of cores kept occupied. This should make parallelism on devices up to some 100 cores practical.

- A considerable investment in training for existing and new developers will be required.

## Acknowledgements

The authors of the report would like to acknowledge helpful input and discussions with Will Buttinger, Dmitry Emeliyanov, Benedikt Hegner, Nils Erik Krumnack, Walter Lampl, Rolf Seuster, Scott Snyder, Vakho Tsulaia, Peter van Gemmeren.

## References

[1] M. Clemencic, B. Hegner, P. Mato, and D. Piparo, Journal of Physics: Conference Series **513** no. 5, (2014) 052028. http://stacks.iop.org/1742-6596/513/i=5/a=052028.

[2] B. Hegner, P. Mato, and D. Piparo, Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2012 IEEE (2012) 2003–2007.

[3] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, Commun. ACM **55** no. 4, (2012) 55–63. http://doi.acm.org/10.1145/2133806.2133822.

[4] Mato, P., et al,. https://inspirehep.net/record/928960/files/lhcb-98-064.pdf.

[5] G. Barrand, I. Belyaev, P. Binko, M. Cattaneo, R. Chytracek, et al., Comput.Phys.Commun. **140** (2001) 45–55.

[6] P. Calafiura, C. G. Leggett, D. R. Quarrie, H. Ma, and S. Rajagopalan, CoRR **cs.SE/0306089** (2003). http://arxiv.org/abs/cs.SE/0306089.

[7] J. Haller, R. Stamen, G. Comune, and C. Schiavi, Tech. Rep. ATL-COM-DAQ-2006-023, CERN, Geneva, Apr, 2006.

[8] N. Berger, T. Bold, T. Eifert, G. Fischer, S. George, J. Haller, A. Höcker, J. Masik, M. Zur Nedden, V. Pérez-Réale, C. Risler, C. Schiavi, J. Stelzer, and X. Wu, Tech. Rep. ATL-COM-DAQ-2007-020, CERN, Geneva, Jun, 2007. Poster presented at RT07. Paper submitted to IEEE Transactions on Nuclear Science (TNS).

[9] ATLAS,, "HLT Steering twiki." https://twiki.cern.ch/twiki/bin/viewauth/Atlas/HltSteering.

[10] *LS2 and LS3: The largest Challenges we Know Today*. 2014. https://indico.cern.ch/event/315626/session/2/contribution/54/material/slides/1.pdf.