Implementation of the ATLAS trigger within the multi-threaded software framework AthenaMT

B. Wynne
On behalf of the ATLAS Collaboration

11/10/16

Introduction

As the LHC luminosity increases to 3x its design value in 2021, and 7.5x in 2026, the ATLAS High Level Trigger (HLT) will need to be upgraded to handle an input event rate between 4x and 10x the current maximum of 100kHz

Given current trends in hardware, upgrading our HLT computer farm to cope with this increased load will require multi-threaded software with efficient memory sharing
- At present we rely on a multi-process approach

The LHC collisions will also become more complex to reconstruct, with increasing numbers of overlapping proton-proton collisions in a single event

This motivates us to use reconstruction techniques with more stringent signal/background discrimination, typically found in offline algorithms

Please refer to Simon's talk this morning:
https://indico.cern.ch/event/505613/contributions/2227283/

Unify online and offline environments
Introduce multi-threading

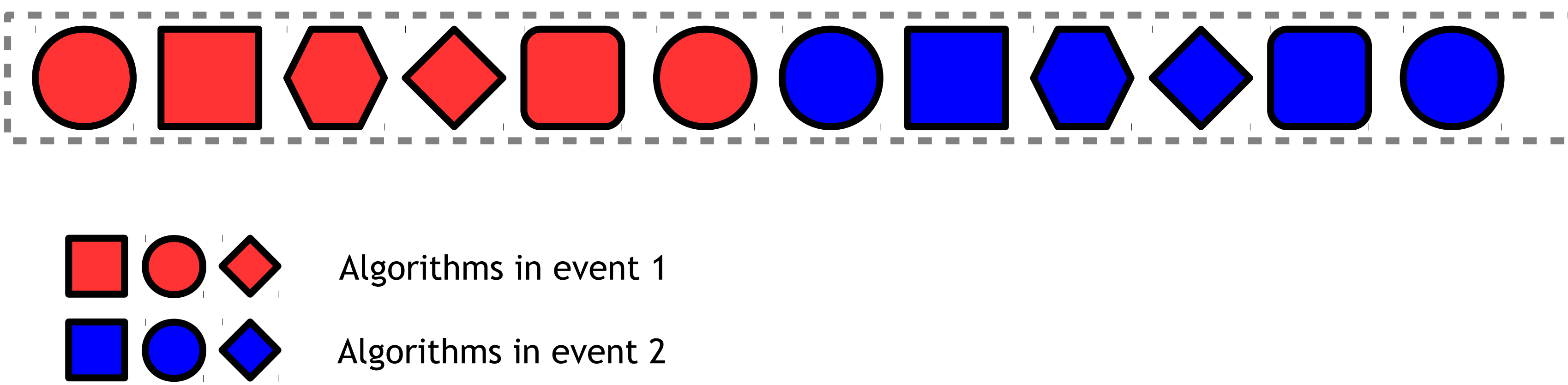
AthenaMT

The ATLAS experiment is developing a new software framework – **AthenaMT** – to introduce multi-threading into the offline reconstruction and HLT workflows

Gaudi → Athena
GaudiHive → AthenaMT

AthenaMT targets intra-event parallelism, by executing individual algorithms in available CPU threads. The same mechanism allows for inter-event parallelism

The crucial requirement is understanding the dependencies between algorithms, to determine when they are safe to execute



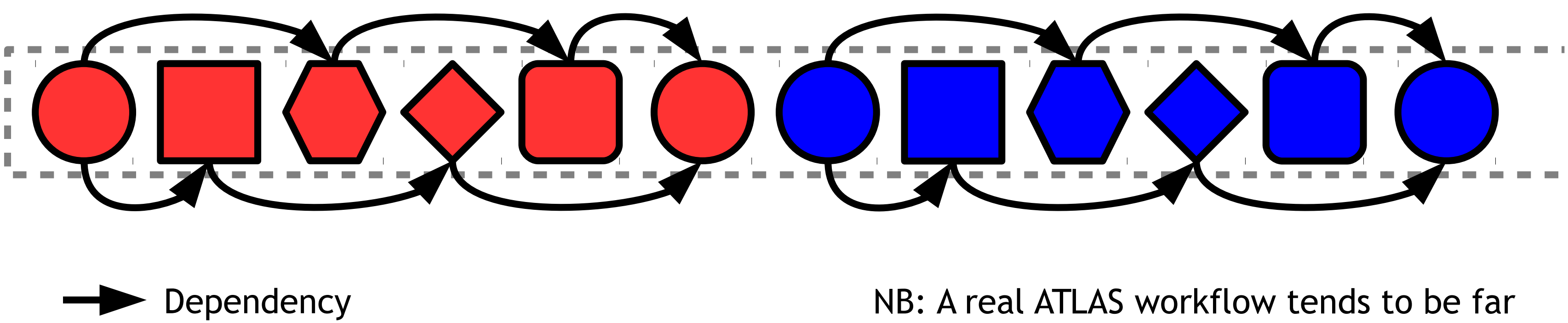
AthenaMT

The ATLAS experiment is developing a new software framework – **AthenaMT** – to introduce multi-threading into the offline reconstruction and HLT workflows

Gaudi → Athena
GaudiHive → AthenaMT

AthenaMT targets intra-event parallelism, by executing individual algorithms in available CPU threads. The same mechanism allows for inter-event parallelism

The crucial requirement is understanding the dependencies between algorithms, to determine when they are safe to execute



NB: A real ATLAS workflow tends to be far more complex. The current HLT includes ~150 algorithms in ~2000 dependency chains

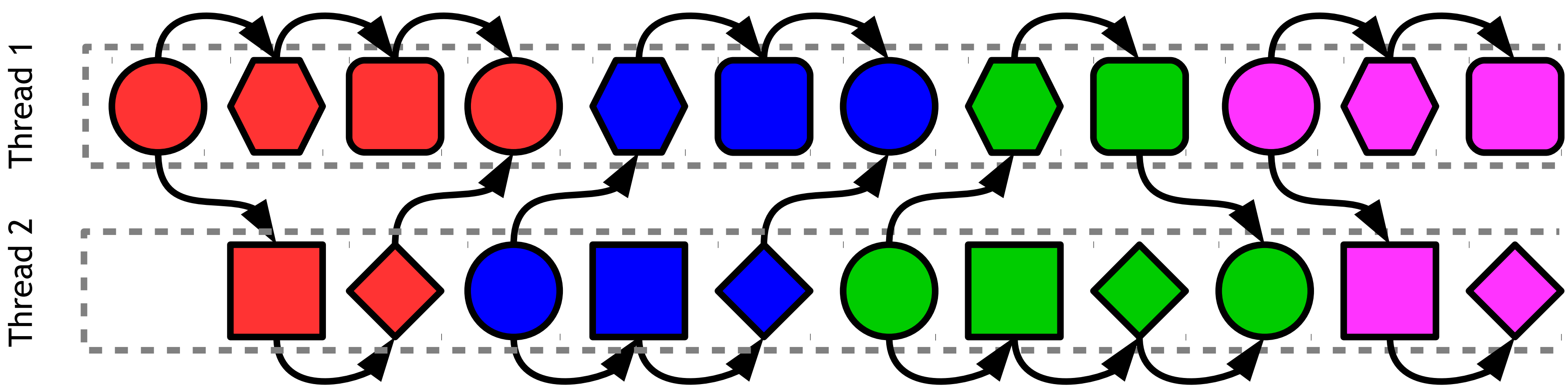
AthenaMT

The ATLAS experiment is developing a new software framework – **AthenaMT** – to introduce multi-threading into the offline reconstruction and HLT workflows

Gaudi → Athena
GaudiHive → AthenaMT

AthenaMT targets intra-event parallelism, by executing individual algorithms in available CPU threads. The same mechanism allows for inter-event parallelism

The crucial requirement is understanding the dependencies between algorithms, to determine when they are safe to execute



Based on slide from G. Stewart

HLT framework comparison

Athena

Single-threaded

HLT-specific "steering" layer
- Schedules algorithms
- Makes trigger decisions

HLT-specific algorithm class for RoI-based reconstruction

AthenaMT

Multi-threaded

Common scheduler for HLT and offline algorithms

Common algorithm class for HLT and offline
- Facilitates code-sharing

Trigger decisions made by menu algorithms

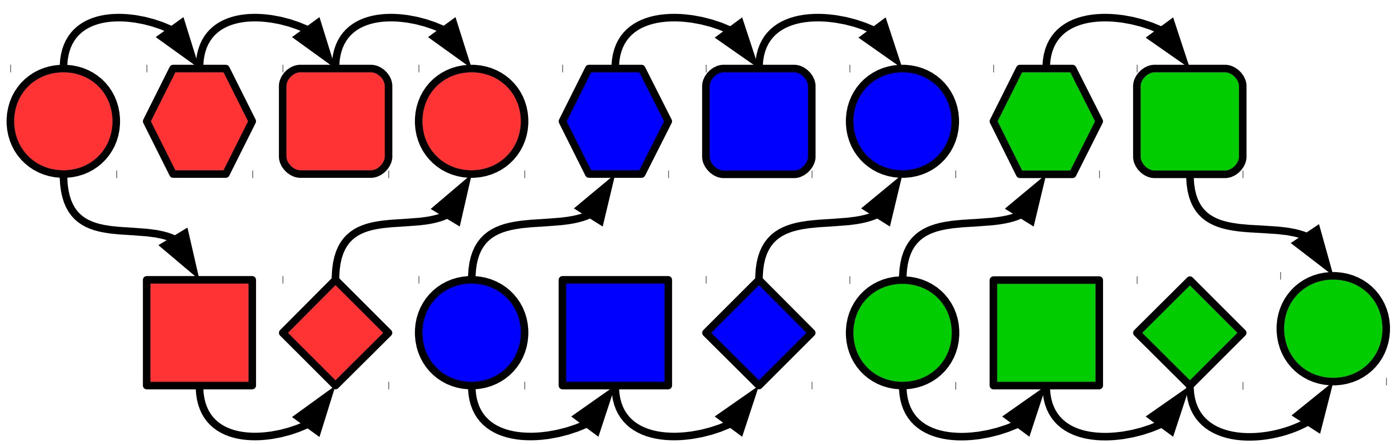
RoI data stored using **EventViews**
- Can be accessed or manipulated by any algorithm
- HLT-specific information stored as event data

Algorithm dependencies

When scheduling algorithms, **AthenaMT** makes the following assumptions

1) If an algorithm is configured, it must be executed for each event
   (i.e. its dependencies are guaranteed to be satisfied at some point)
2) An algorithm is only executed once per event
3) Data dependencies are evaluated for a whole event

As a result, the same graph is executed for each event (ignoring details of thread assignment)



AthenaMT and the trigger

The ATLAS trigger system is based on the concept of Regions of Interest (RoIs)

To reduce the readout bandwidth and CPU time requirements, HLT processing is restricted to small windows in η-φ space identified as containing an energetic/interesting particle

This runs contrary to many of the **AthenaMT** assumptions, which arose from an offline perspective

1) Algorithms must not run if there is no appropriate RoI
2) Algorithms may be executed for multiple RoIs in a single event
3) Data objects are specific to an RoI

In Run 1 and Run 2, this behaviour was implemented using an HLT-specific layer on top of the (offline) **Athena** framework

The goal for **AthenaMT** is to support both offline and HLT processing, to allow sharing of algorithmic code

DataHandles and EventViews

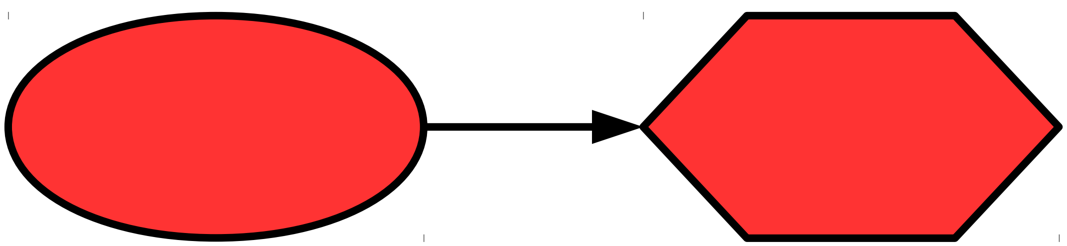
All **AthenaMT** algorithms produce and consume data via smart pointers called **DataHandles**

An algorithm declares **DataHandles** for each input and output data object, specifying the data type and the name it is stored under

The scheduler then examines all **DataHandles** to determine the dependencies between algorithms

During processing of an event, the framework will update each **DataHandle** to point to the location in memory where each data object should be stored

Simplified data dependency cartoon



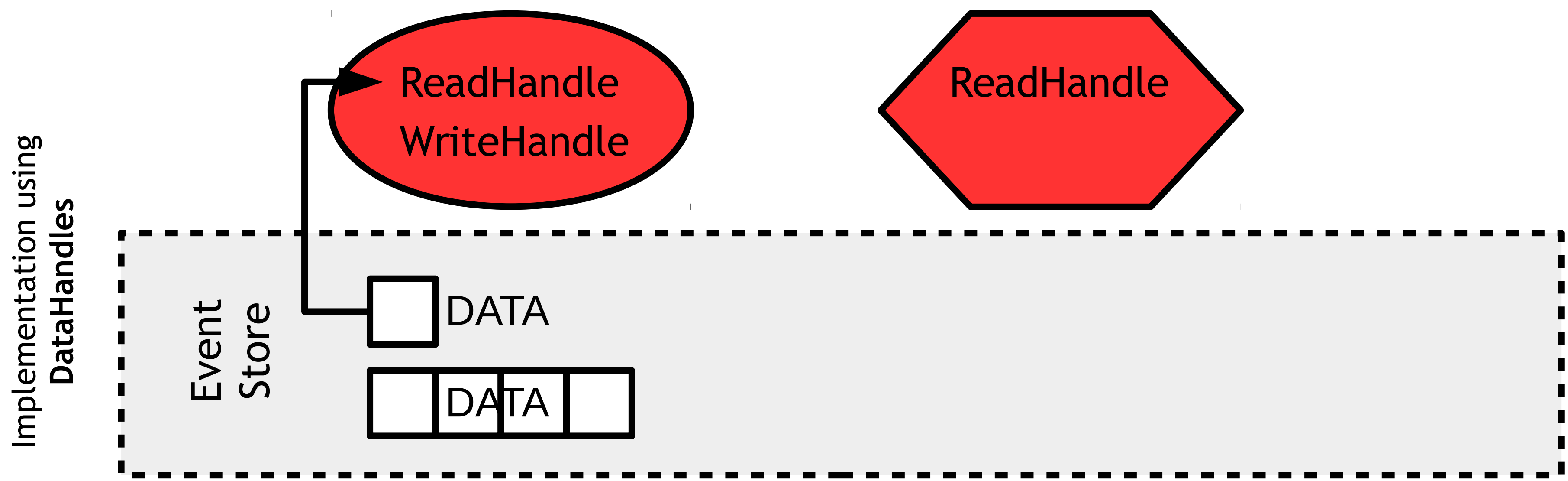
DataHandles and EventViews

All **AthenaMT** algorithms produce and consume data via smart pointers called DataHandles

An algorithm declares **DataHandles** for each input and output data object, specifying the data type and the name it is stored under

The scheduler then examines all **DataHandles** to determine the dependencies between algorithms

During processing of an event, the framework will update each **DataHandle** to point to the location in memory where each data object should be stored



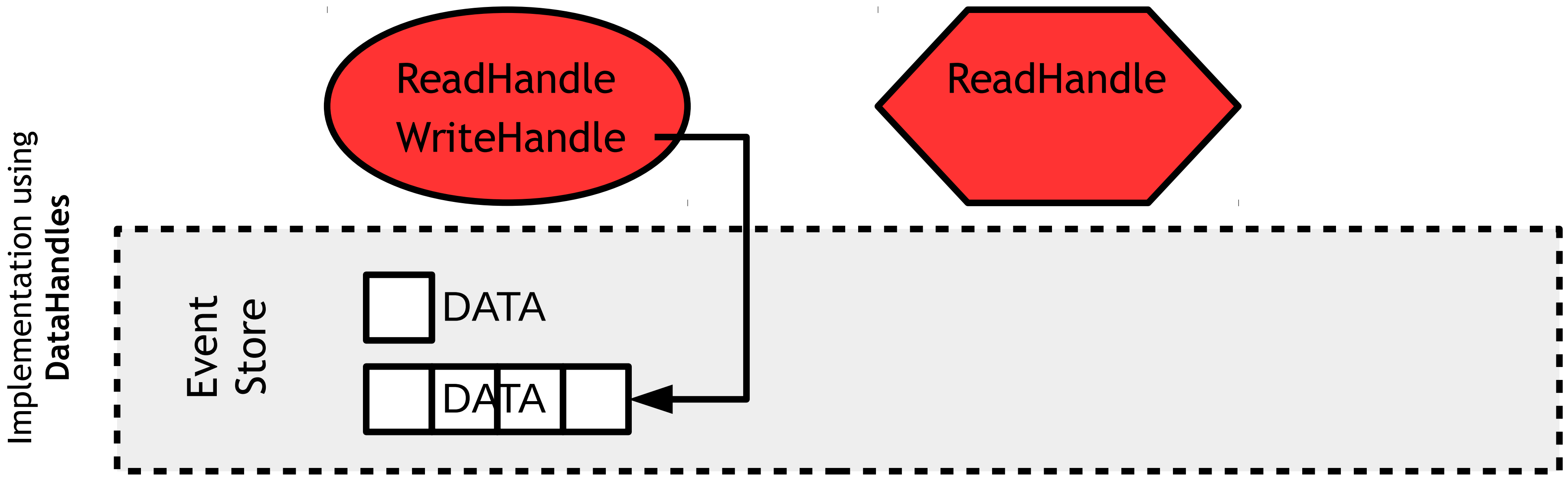
DataHandles and EventViews

All **AthenaMT** algorithms produce and consume data via smart pointers called DataHandles

An algorithm declares **DataHandles** for each input and output data object, specifying the data type and the name it is stored under

The scheduler then examines all **DataHandles** to determine the dependencies between algorithms

During processing of an event, the framework will update each **DataHandle** to point to the location in memory where each data object should be stored



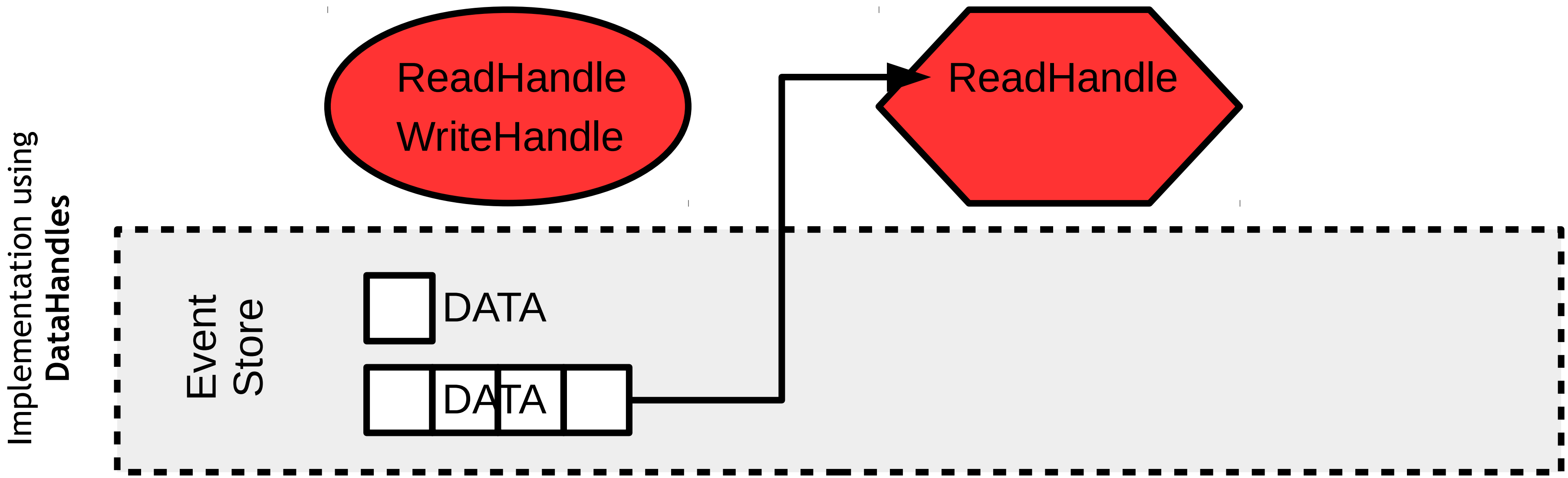
DataHandles and EventViews

All AthenaMT algorithms produce and consume data via smart pointers called **DataHandles**

An algorithm declares **DataHandles** for each input and output data object, specifying the data type and the name it is stored under

The scheduler then examines all **DataHandles** to determine the dependencies between algorithms

During processing of an event, the framework will update each **DataHandle** to point to the location in memory where each data object should be stored

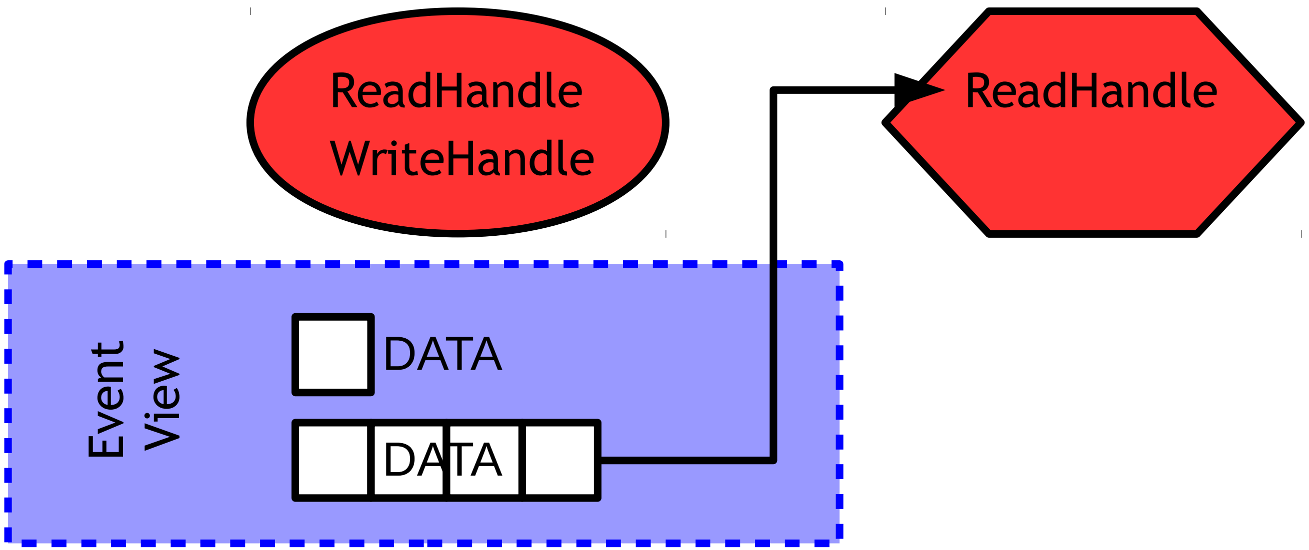


DataHandles and EventViews

An offline algorithm (designed to run on a whole event) can be run unmodified in an RoI created by the trigger, simply by having the framework modify the **DataHandles**

We have implemented an **EventView** class that can be used interchangeably with the whole event store. Each view is populated with data corresponding to a single RoI, and then connected to an algorithm via the **DataHandles**

EventViews are treated like a standard data object, so can be created or manipulated by any algorithm

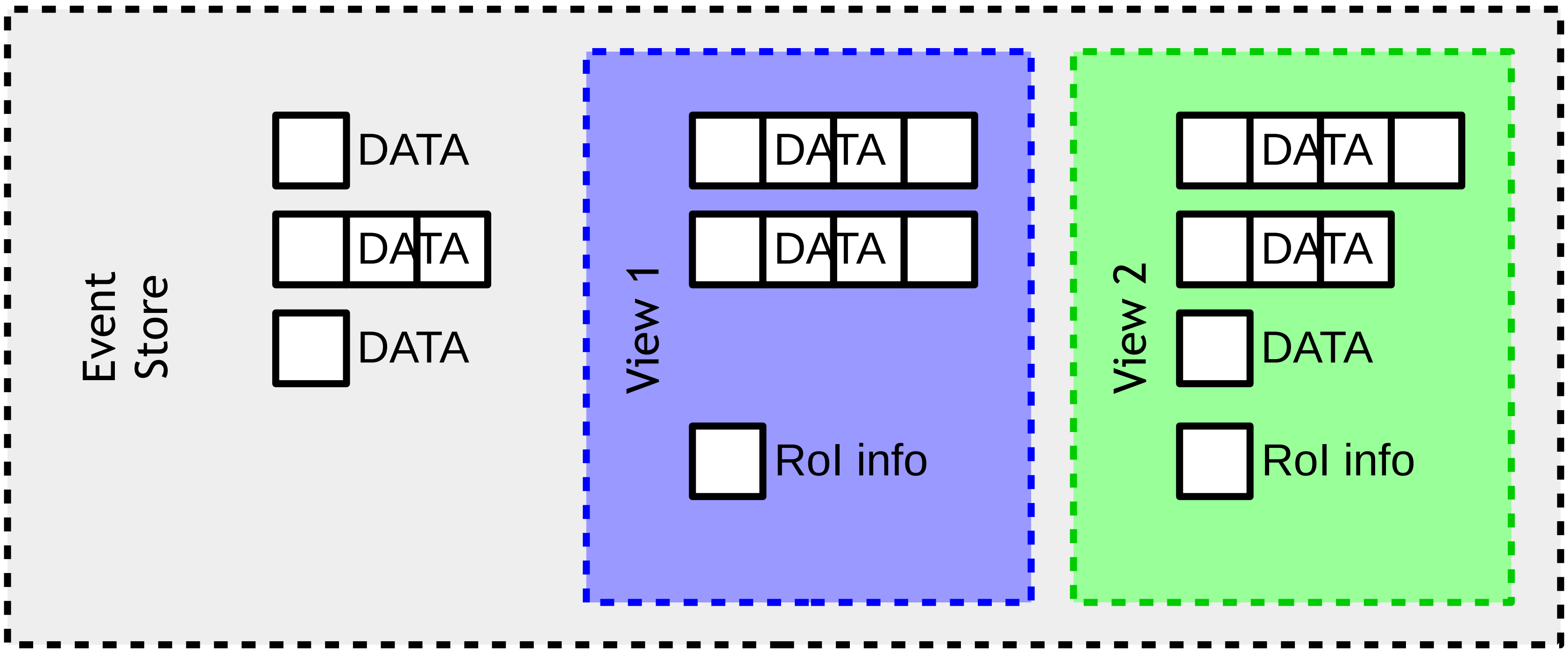


EventViews

Each **EventView** implements the same interface as the whole event store, and presents a subset of the data it contains

The views are intended to be general-purpose objects

- They can contain data objects that describe a corresponding RoI
- Allows for potential alternative use-cases



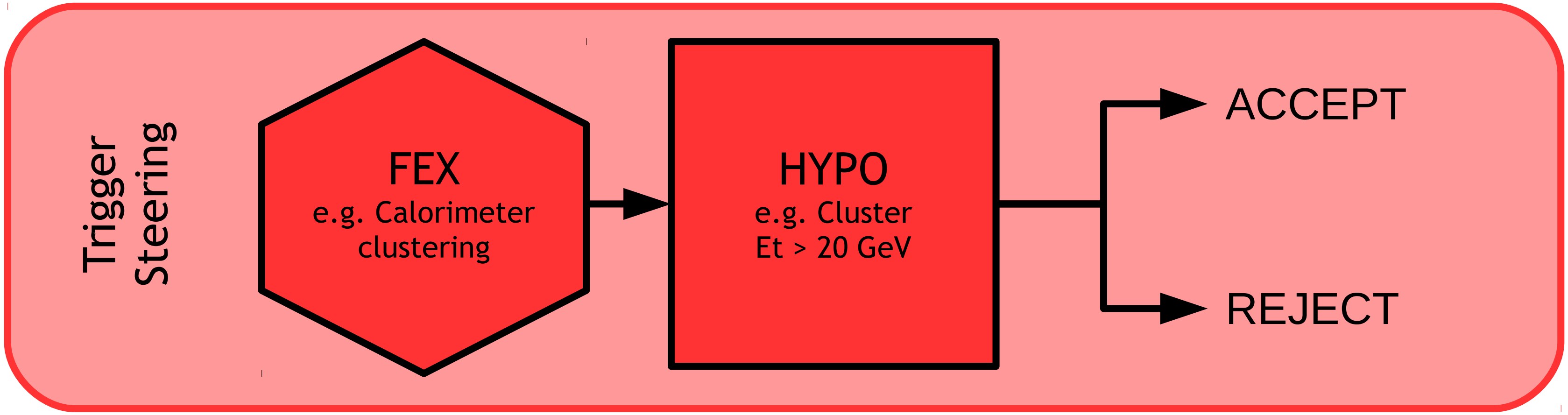
Trigger menu

In the HLT-specific layer of the old **Athena** framework, the scheduling of algorithms and resulting event accept/reject decisions were made by the "steering" class

The **AthenaMT** scheduler replaces steering, but does not take trigger decisions

A decision is made in three stages:
1) Feature EXtraction (FEX) algorithms reconstruct detector data
2) Hypothesis (HYPO) algorithms apply selection criteria
3) Passed/failed hypotheses compared to trigger "menu" to select events

The first two stages were handled by algorithms, the third by the steering itself



Trigger menu

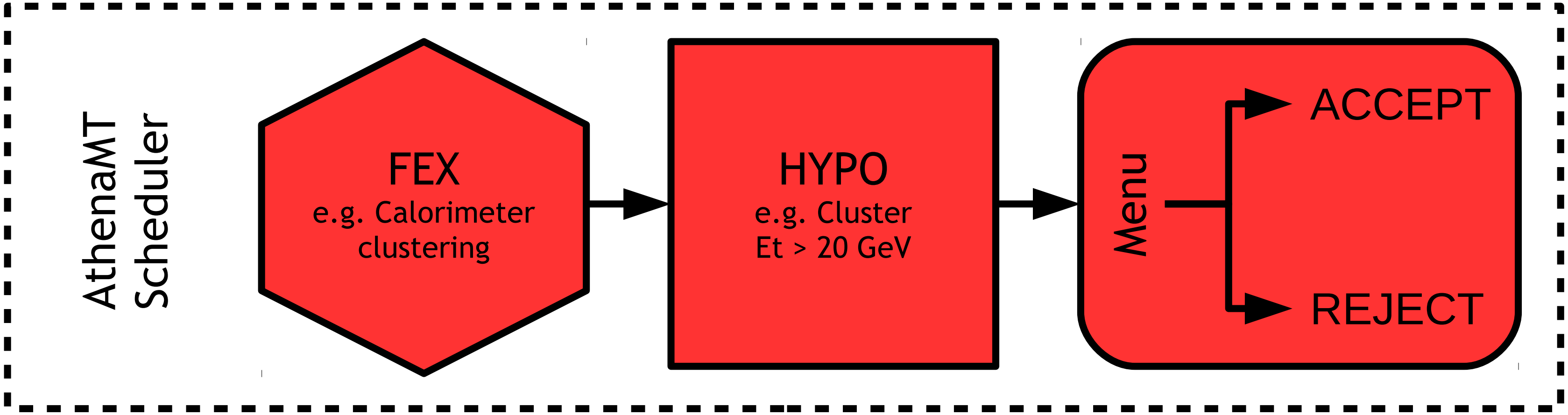
In the HLT-specific layer of the old **Athena** framework, the scheduling of algorithms and resulting event accept/reject decisions were made by the "steering" class

The **AthenaMT** scheduler replaces steering, but does not take trigger decisions

A decision is made in three stages:
1) Feature EXtraction (FEX) algorithms reconstruct detector data
2) Hypothesis (HYPO) algorithms apply selection criteria
3) Passed/failed hypotheses compared to trigger "menu" to select events

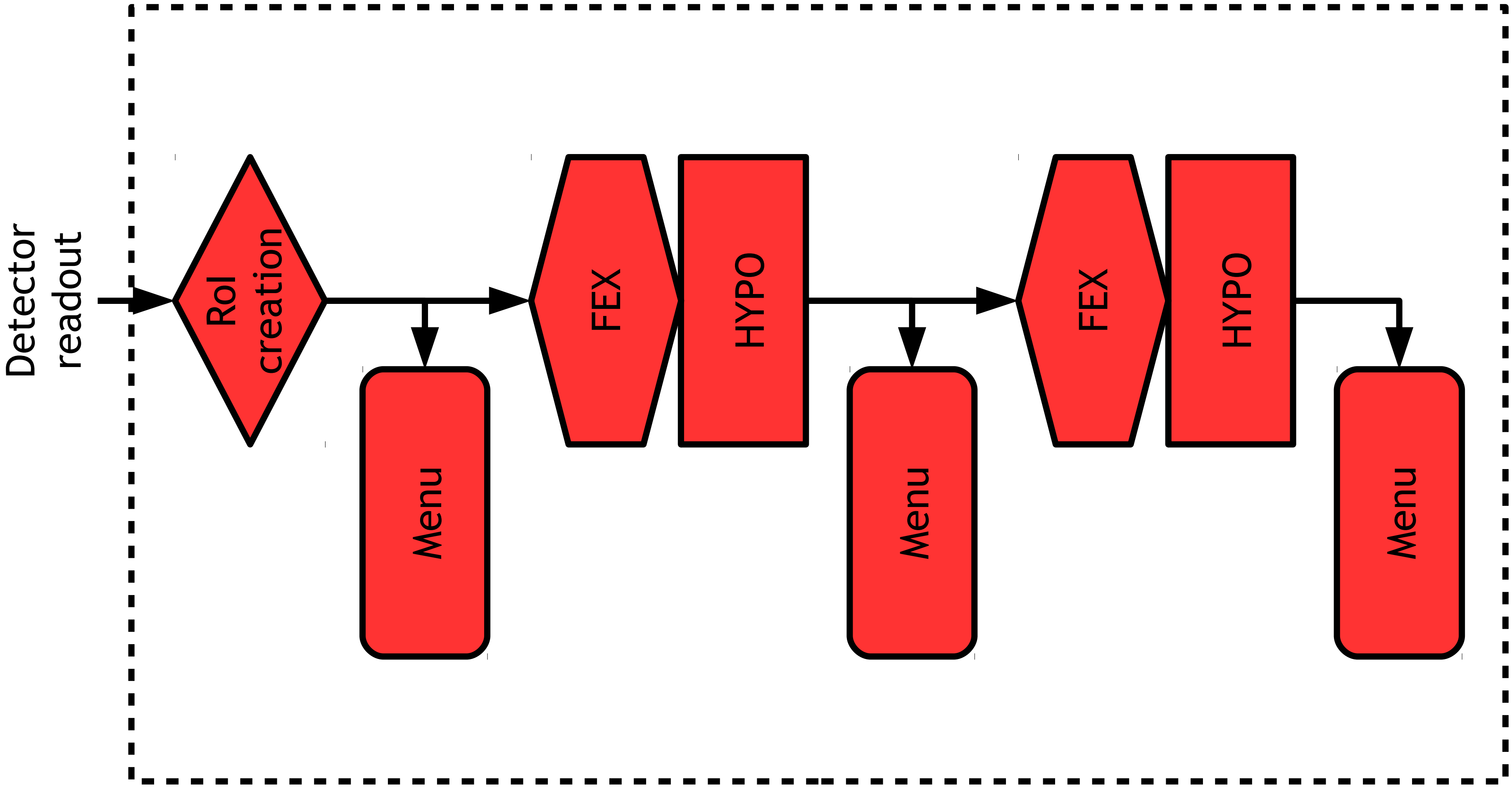
The first two stages were handled by algorithms, the third by the steering itself

We now introduce menu algorithms, fully replacing the steering



Menu algorithms

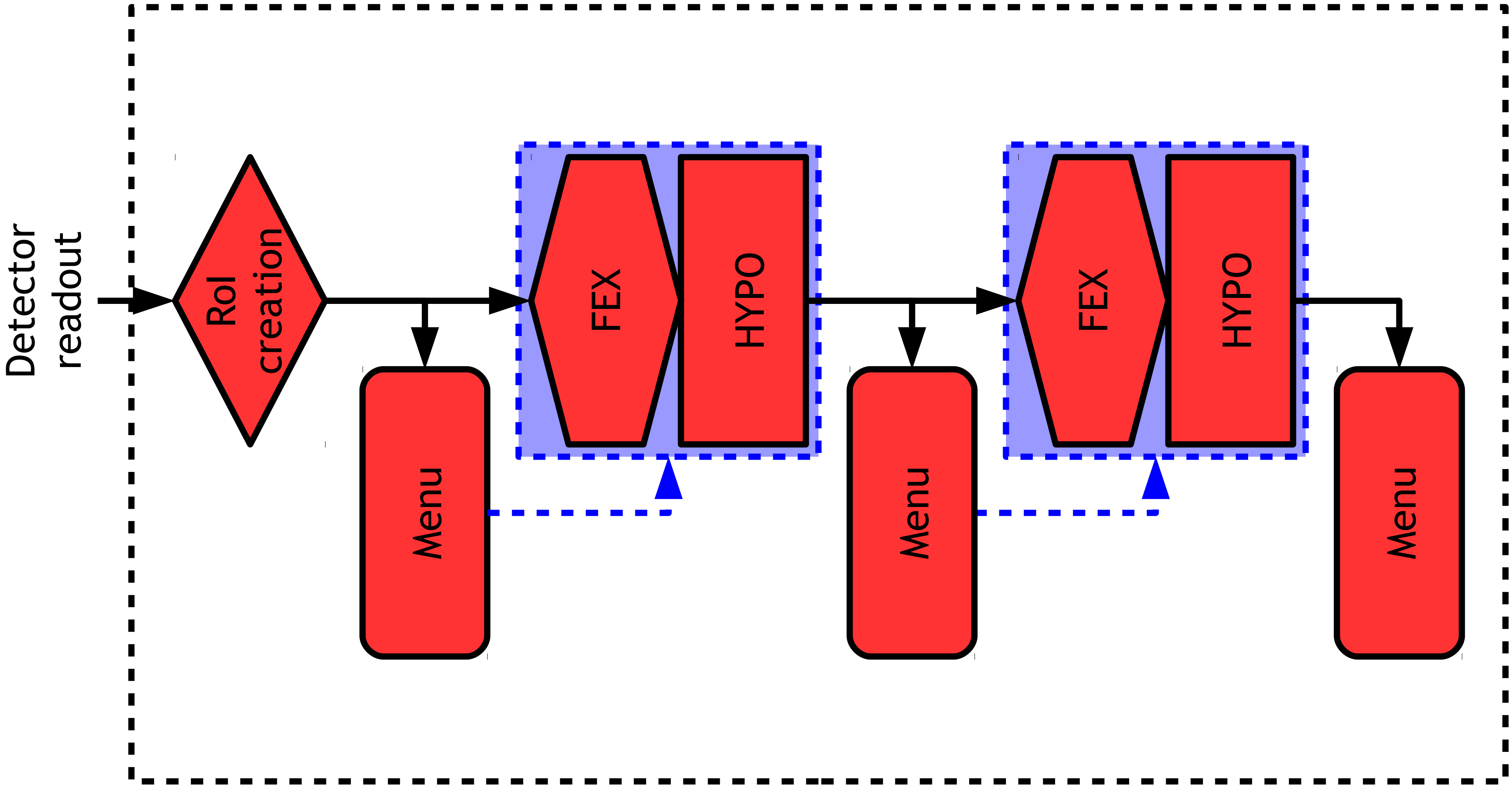
To provide early rejection, menu decisions are taken in several stages, with FEX and HYPO algorithms scheduled in between



Menu algorithms

RoI information is read in from the detector and used to create and populate **EventViews**

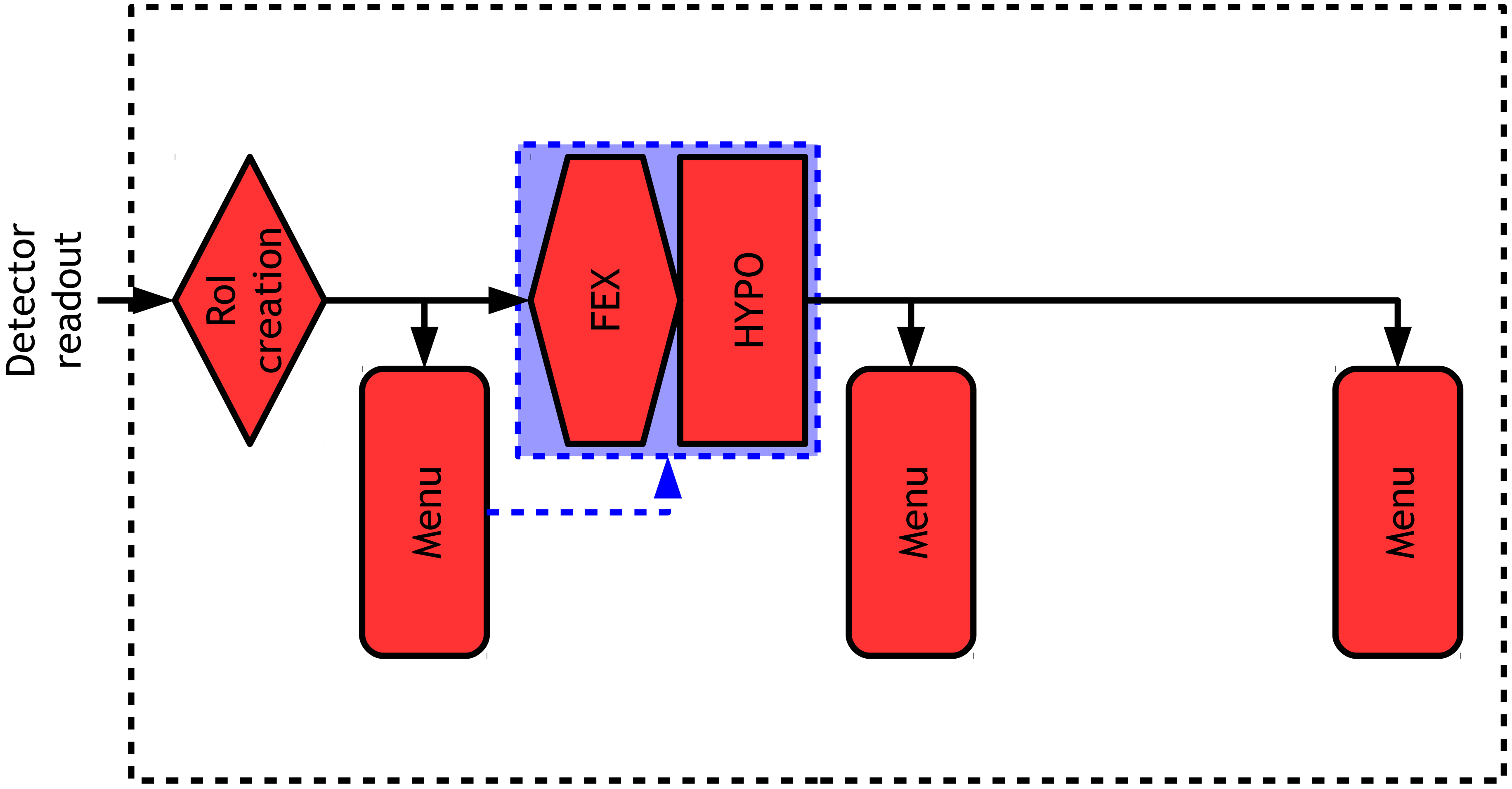
The menu algorithms are responsible for assigning FEX and HYPO algorithms to an appropriate view



Menu algorithms

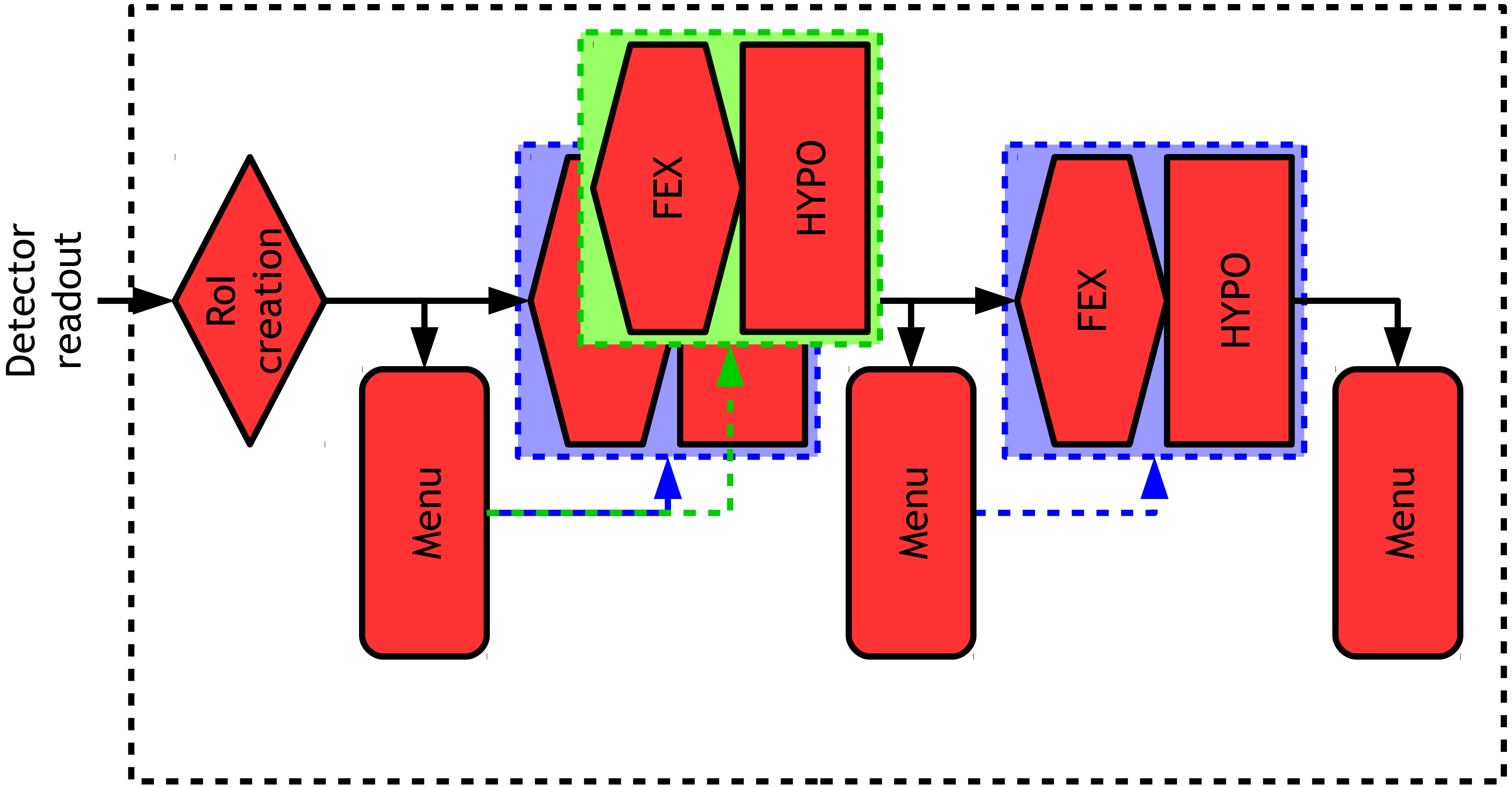
If there is no appropriate RoI, or if a decision is taken to reject the event, then FEX and HYPO algorithms are never scheduled

FEX and HYPO algorithms are configured never to run on a whole event, and are skipped by the **AthenaMT** scheduler



Menu algorithms

The menu algorithms prompt the scheduling of FEX and HYPO algorithms, allowing them to be run multiple times per event where there are multiple RoIs to process



HLT framework comparison

Athena

Single-threaded

HLT-specific "steering" layer
- Schedules algorithms
- Makes trigger decisions

HLT-specific algorithm class for RoI-based reconstruction

AthenaMT

Multi-threaded

Common scheduler for HLT and offline algorithms

Common algorithm class for HLT and offline
- Facilitates code-sharing

Trigger decisions made by menu algorithms

RoI data stored using **EventViews**
- Can be accessed or manipulated by any algorithm
- HLT-specific information stored as event data

Status and plans

All components of the **AthenaMT** HLT workflow described here have a working prototype

Currently being combined in a demonstrator for the full workflow
- Menu algorithm workflow exists with dummy algorithms
- Migrating real algorithms to use **DataHandles**
- Adding **EventView** manipulation to the menu algorithms

Aiming for a first implementation using a limited set of algorithms by the end of this year

Will add algorithms to the menu as they are migrated over the following years, with the plan to have a complete implementation ready for 2019

Currently the HLT uses
~150 algorithms (connected by ~2000 “chains” defining data flow and selection)
~400 tools (configurable sub-components of algorithms)
~100 services (globally accessible interfaces to I/O, configuration, etc.)

Some components will be simplified or replaced, but the rest must be migrated

BACKUP

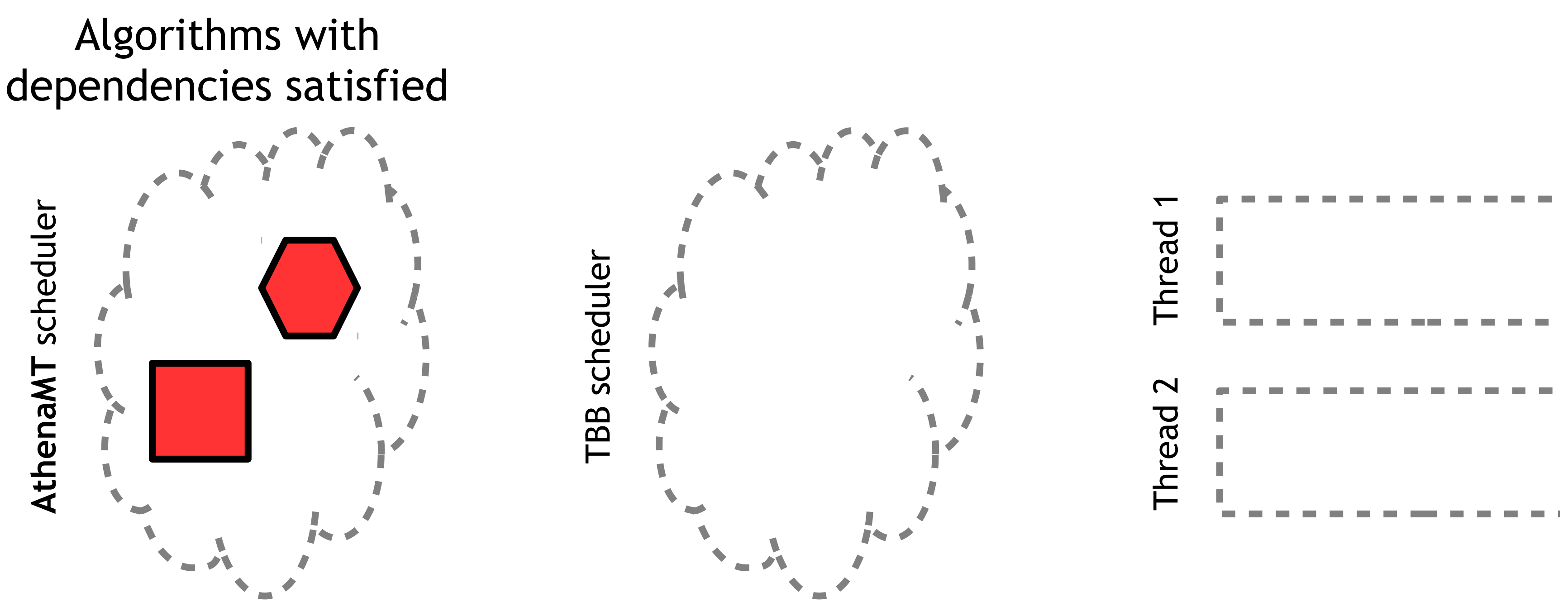
Intel Thread Building Blocks

The **GaudiHive/AthenaMT** scheduler relies on assigning a unit of work – an algorithm - to a CPU thread as it becomes free

Intel Thread Building Blocks (TBB) is the mechanism used to achieve this

When an algorithm's dependencies are satisfied, it is wrapped in a TBB task class by the **AthenaMT** scheduler

The task is then assigned to a thread by the TBB scheduler



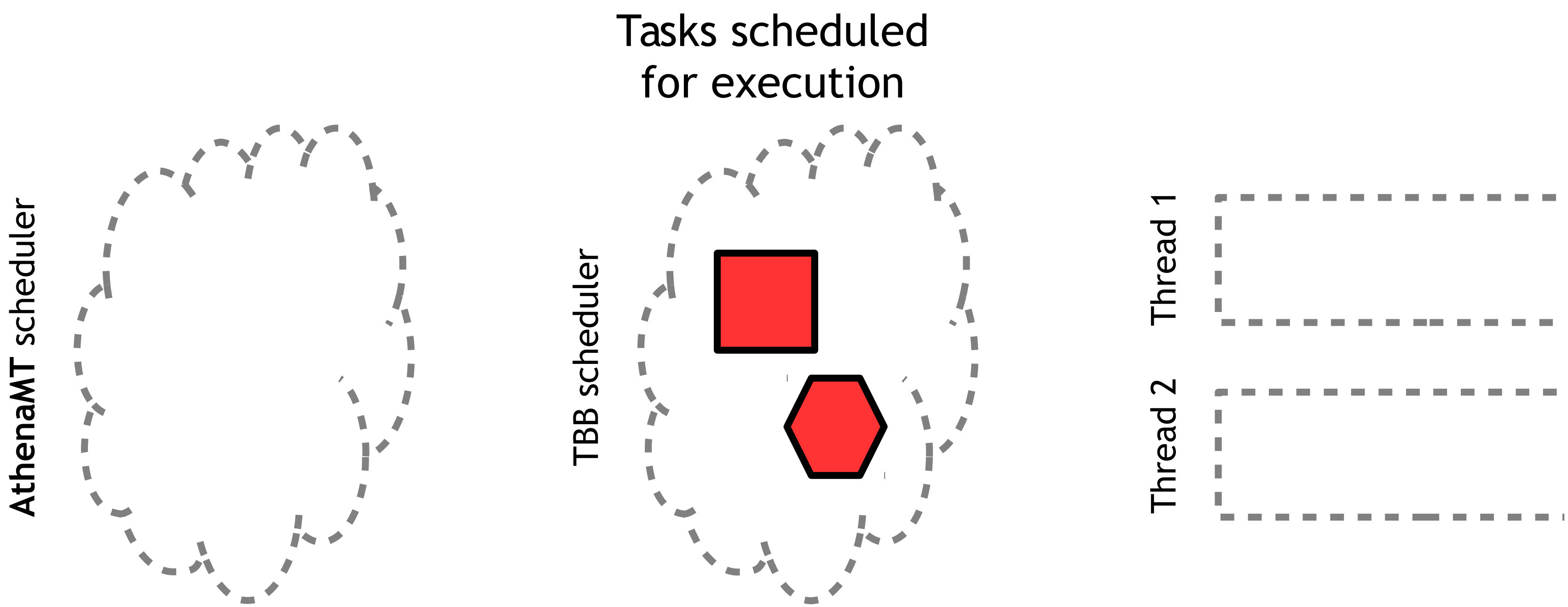
Intel Thread Building Blocks

The **GaudiHive/AthenaMT** scheduler relies on assigning a unit of work – an algorithm - to a CPU thread as it becomes free

Intel Thread Building Blocks (TBB) is the mechanism used to achieve this

When an algorithm's dependencies are satisfied, it is wrapped in a TBB task class by the **AthenaMT** scheduler

The task is then assigned to a thread by the TBB scheduler



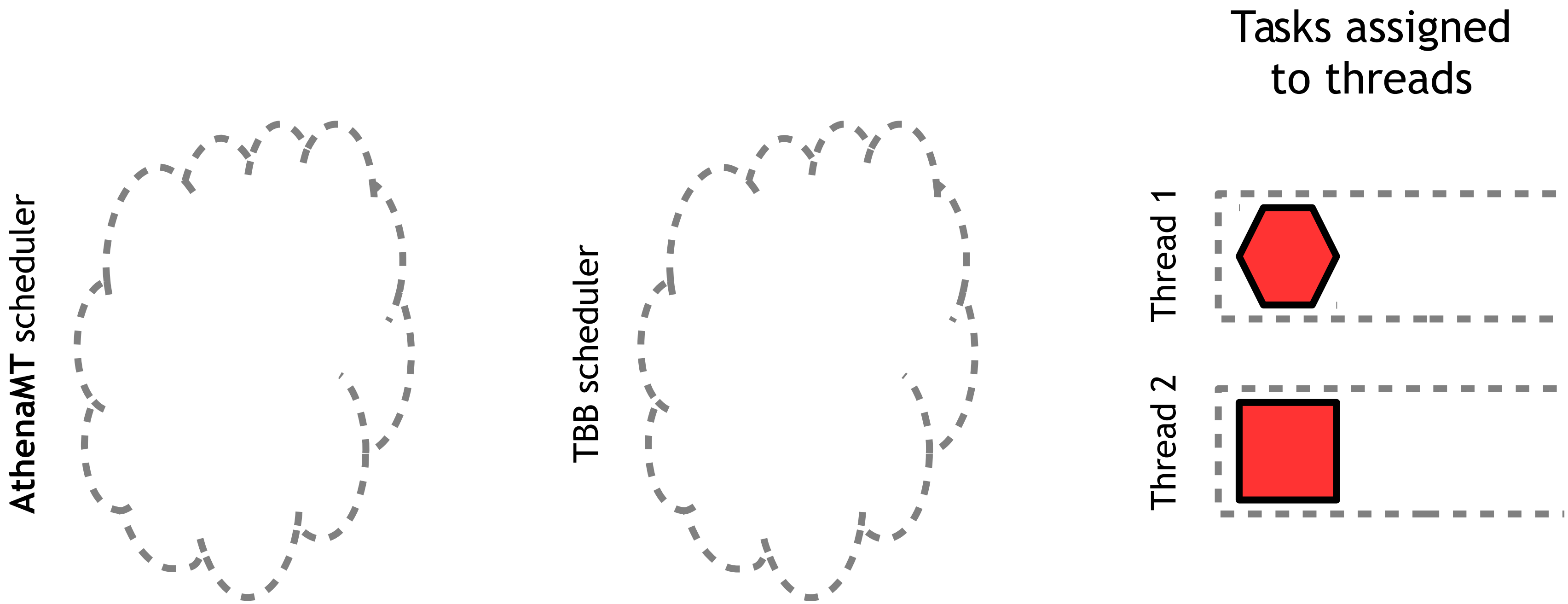
Intel Thread Building Blocks

The **GaudiHive/AthenaMT** scheduler relies on assigning a unit of work – an algorithm - to a CPU thread as it becomes free

Intel Thread Building Blocks (TBB) is the mechanism used to achieve this

When an algorithm's dependencies are satisfied, it is wrapped in a TBB task class by the **AthenaMT** scheduler

The task is then assigned to a thread by the TBB scheduler



Intel Thread Building Blocks

The **GaudiHive/AthenaMT** scheduler relies on assigning a unit of work – an algorithm - to a CPU thread as it becomes free

Intel Thread Building Blocks (TBB) is the mechanism used to achieve this

When an algorithm's dependencies are satisfied, it is wrapped in a TBB task class by the **AthenaMT** scheduler

The task is then assigned to a thread by the TBB scheduler

- We don't use the mechanisms in TBB for resolving algorithm dependency graphs, it's just used to assign tasks to threads

- One important feature of TBB that we do use: tasks can create other tasks, allowing for internal parallelism in our algorithms

Menu algorithms

Maybe say something about HypoTools versus many RoIs in Run1?