ISOTDAQ

International School of Trigger and Data Acquisition

Design and Implementation of a Monitoring System

Serguei Kolos,

University of California, Irvine

What you are expected to learn in the next hour

- ❑ Why systems need to be monitored
- ❑ A little bit of theory
- ❑ The Basic one-size-fits-all Architecture
  - ❑ Technology independent
- ❑ Implementation Strategy:
  - ❑ With a few technology examples
- ❑ Data Quality Monitoring

Why systems need to be Monitored?

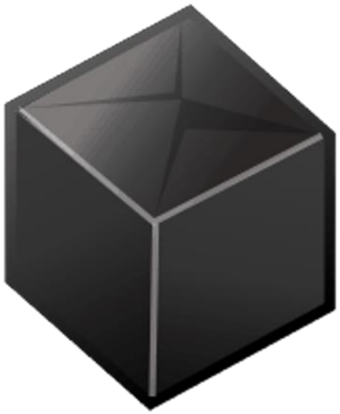
- Cos the Universe is not Perfect
- The rate of failures is proportional to the system complexity
- Monitoring is indispensable for the System control

How Higgs boson discovery would have looked like in an ideal world

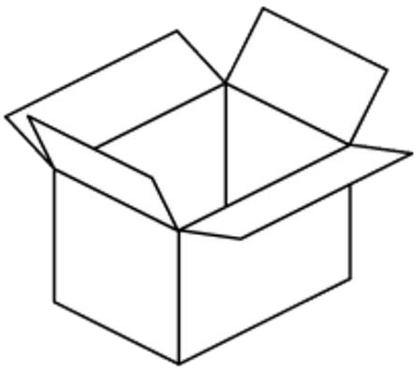
What happens in reality

- A complex project has a chance to success only if it is ready to deal with problems
- Monitoring System provides the first line of defense:
  - Detects, Reports, Helps to Investigate

Two Main Approaches for Monitoring

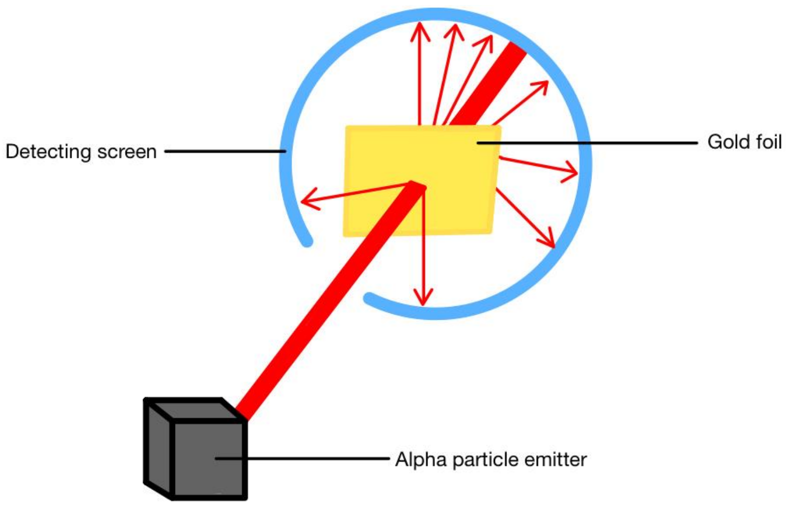


Black Box	White Box
Passive	Active
Polling	Notification
Synchronous	Asynchronous



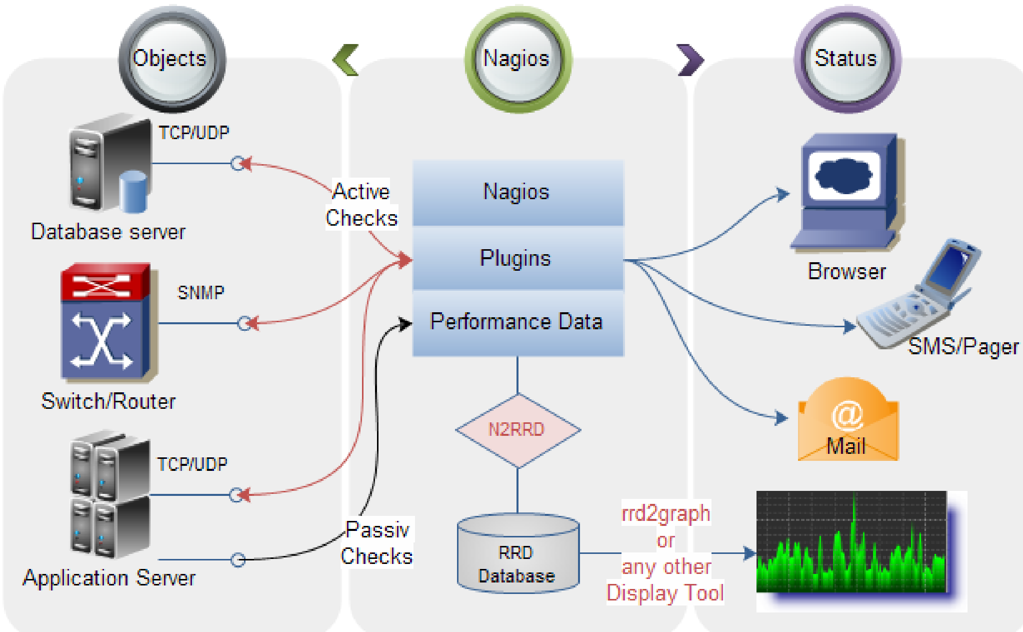
The **Black Box** Monitoring Approach

Rutherford gold foil experiment



- System to be monitored is a ***Black Box***
- Use well-defined procedures as probes for the ***Black Box*** and measure the result

The **Black Box** Monitoring Example



- [Nagios]() is a classical example of the ***Black Box*** monitoring
  - Provides checks for commodity HW and SW
  - Allows to integrate custom checks
- Other examples: [Icinga](), [Ganglia](), etc.

Black Box Approach for DAQ system?

- **Data AcQuisition** is an heterogeneous field
  - Boundaries not well defined
  - An **alchemy** of physics, electronics, networking, computer science, …
  - Hacking and experience
  - …, money and manpower matter as well

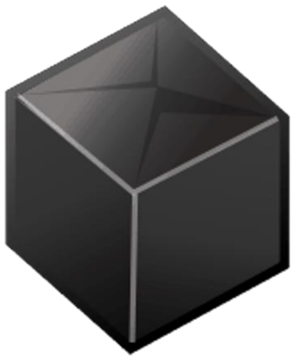
- DAQ system components operate at high rate:
  - Polling for monitoring information is inefficient
- A DAQ system has many **custom** HW and SW:
  - Good opportunity to do monitoring in a better way…

What if the Universe was created by a Computer Scientist?

The White Box Approach

- Objects expose information about their states:
  - E.g. coordinates and velocities of the particles
- Physicists would merely take care of visualizing this information

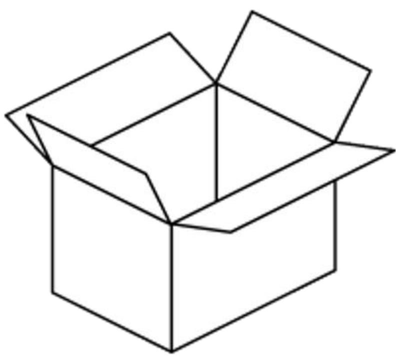
Off-the-shelf Software Solutions?



Plenty of ready-made solutions

Only applies to commodity HW/SW

Custom components require development of custom probes



No ready-made solutions

Can be constructed using:

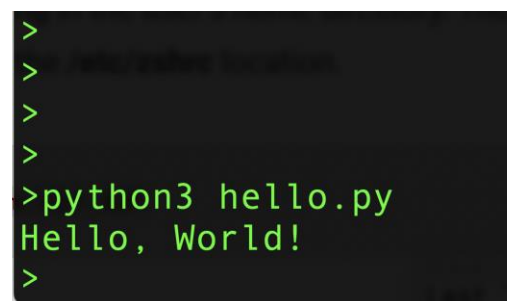
- Commodity tools and libraries
- A little bit of custom programming
- <u>Some good recipes</u>

The Simplest “White Box” Example

The monitoring API function

The message

```
print(“Hello, World”)
```

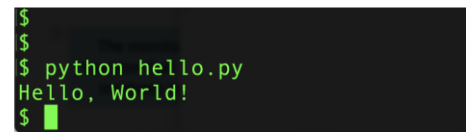


The Basic Architecture

API → Communication → Visualisation

print()

Python builtins module



- **API**
  - **The critical component**
  - DAQ system see only this **API**
  - It must be independent of the **Communication** and **Visualisation**
- **This is not the case!**
  - **Communication** and **Visualisation** are strictly bound to the print() function API

Another Issue with the print() function

print("Hello, World")

- Not bad for a single application but doesn't scale
- With multiple applications running for a long time, we want to know:
  - **When** did something happen?
  - **Where** did it come from?
  - **How important** is it?
- Do better solutions exist?

Logging API to the rescue

```python
import logging
logging.basicConfig(level=logging.INFO,
 format="%(asctime)s %(levelname)s\
 [%(filename)s:%(lineno)s%(funcName)s()] %(message)s")

logging.info("Hello, World!")
```

Use the standard well-designed API

The output format can be easily customized

```
$
$
$ python hello.py
2022-06-14 14:57:37,493 INFO [hello.py:5<module>()] Hello, World!
$
```

Timestamp *Severity* *Origin*

The Basic Architecture

- The **Logging API** and **Communication** layers are fully independent
- **The Logging API**
  - Well-designed and mature
- **Communication:**
  - Different implementations exist on the market
  - Can be exchanged transparently for the end-user applications

Programming Languages Support

Python

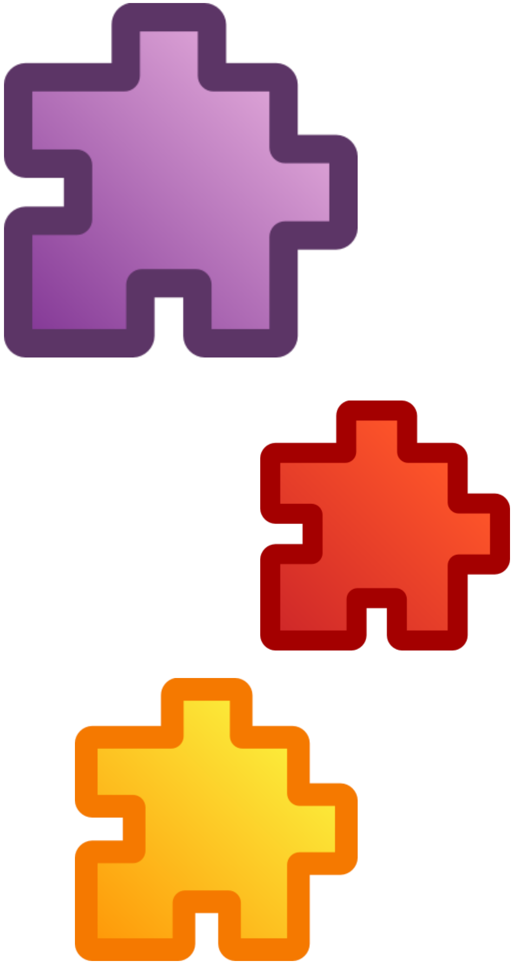
```python
import logging

class Logger:
  def critical(msg, *args, **kwargs):
  def debug(msg, *args, **kwargs):
  def error(msg, *args, **kwargs):
  def info(msg, *args, **kwargs):
  def warning(msg, *args, **kwargs):
```

Java

```java
import java.util.logging.Logger

class Logger {
  void severe(String msg);
  void fine(String msg);
  void error(String msg);
  void info(String msg);
  void warning(String msg);
}
```

Existing Appenders for the Java Logging API

- **CassandraAppender** - writes its output to an Apache Cassandra database
- **FileAppender** – writes events to an arbitrary file.
- **FlumeAppender** - Apache Flume is a distributed, reliable and highly available system for efficiently collecting, aggregating, and moving large amounts of log data
- **JDBCAppender** - writes log events to a relational database table using standard JDBC
- **NoSQLAppender** - writes log events to a NoSQL database
- **SMTPAppender -** sends an e-mail when a specific logging event occurs, typically on errors or fatal errors
- **ZeroMQAppender -** uses the JeroMQ library to send log events to one or more ZeroMQ endpoints



What about C++?

- Rare case where using MACRO for the public API is a viable option

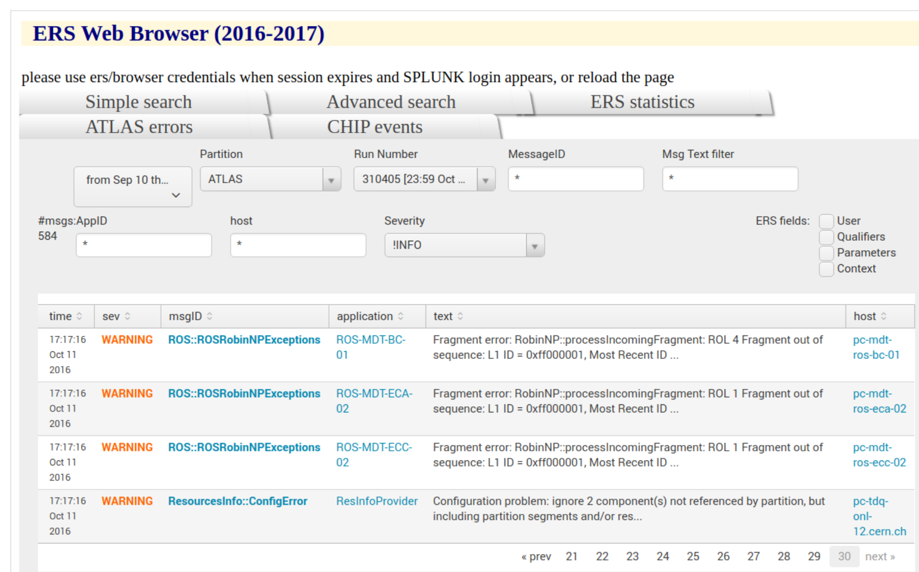
```
DAQ_LOG_CRITICAL("File '" << file_name << "' not found")
DAQ_LOG_ERROR(…)
DAQ_LOG_WARNING(…)
DAQ_LOG_INFO(…)
DAQ_LOG_DEBUG(…)
```

- Initial implementation may be trivial:

```
#define DAQ_LOG_CRITICAL(m) std::cerr << m << std::endl;
```

- A scalable implementation can be provided later:
  - Will not affect users' code

The ATLAS Experiment: Error Reporting System

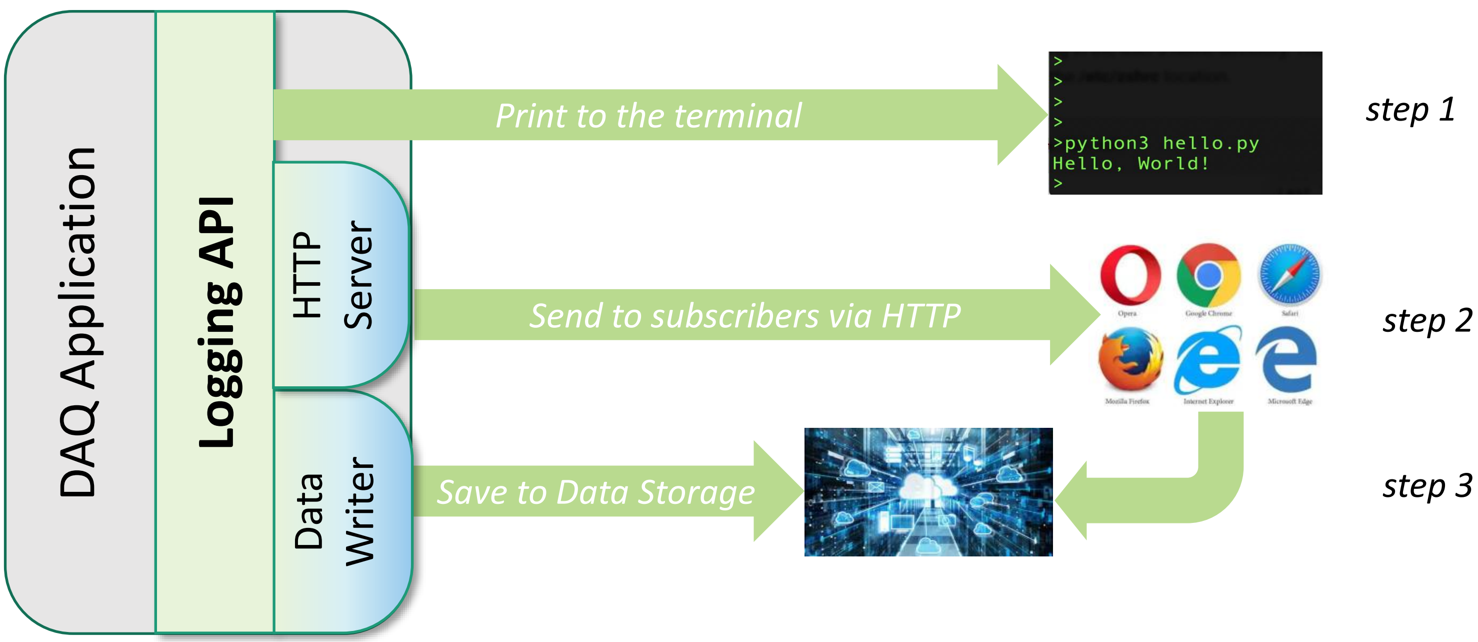


[1] Common Object Request Broker Architecture – inter-process communication technology

[2] Splunk – A software platform to stream and collect data

Evolving the Monitoring System Implementation

- The destination of the messages can be changed at any moment:
  - No changes in the Software Applications required!
- Data Storage is optional but very handy:
  - Adds **persistence** – can be used for postmortem analysis



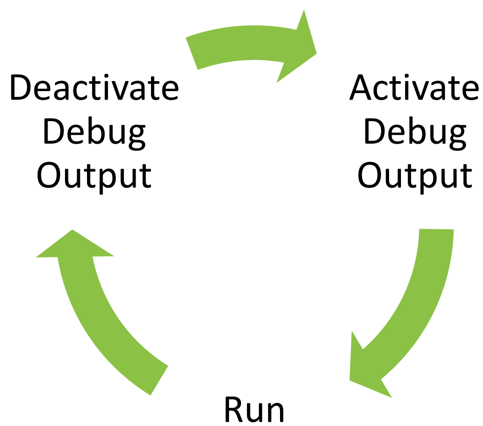
Set Priorities Properly

- Choose (or implement) the Monitoring **API** <u>before starting to implement the DAQ system</u>:
  - The Monitoring must be used by all components of the DAQ system
  - Changing them later will be a pain
- Can take care about **Communication** and **Visualization** implementations later:
  - Using simple output to terminal would be sufficient for the beginning

- Advantages:
  - Using the monitoring system will exercise its functionality and performance
  - Learn the best ways of presenting information
  - Speed up the DAQ system development

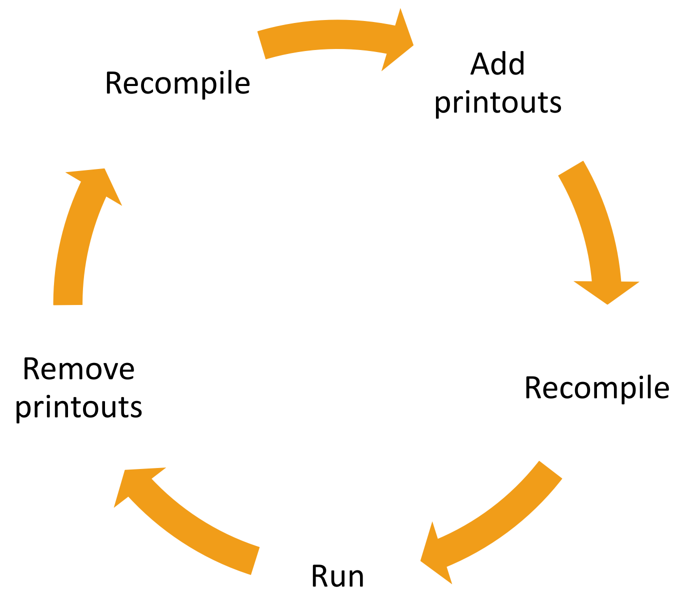
How Monitoring System can speed up DAQ System Development

Efficient debugging cycle using Monitoring API



- *Reduces time for debugging*
- *Optimizes the placement of DEBUG output in the code*

"Traditional" debugging cycle



Can we Extend the Same Ideas to the Other Types of Monitoring Data?

Monitoring Data Types

- **Messages** – used to inform about anything of importance that happens in the system
- **Metrics** – show how the system performs:
    - Values of properties of the software and hardware system components

Main Metrics Types

Counter

- Monotonically increasing integer number
- Simple to monitor:
  - Last value for the last time period
- Examples:
  - Cumulative totals: number of triggers, number of bytes sent/received, etc.

Gauge

- Arbitrary changing value:
  - Integer or floating point
- Monitoring can be tricky:
  - Last value
  - Mean value
  - Min/Max values
- Examples:
  - Resources usage: CPU, memory, buffe
  - Rates: triggers/s, bytes/s, etc.
  - HW Properties: voltage, current, temperature, etc.

Metrics Monitoring Requirements

- ✓Must be displayed as time series
- ✓Must be accessible in real-time
- ✓Must be recorded to be checked later

Reusing the Same Architecture

- **API must be independent** of the **Communication** and **Visualisation**
- **Communication** and **Visualisation** may be changed many times during the project life-time

A Common API for Metrics?

- There is no commonly accepted API for Metrics:
  - SW tools for metrics collection and analysis use their proprietary APIs
- This may not be a problem for a small short-living project:
  - Directly using a specific SW API is a viable option
  - Be careful to choose a SW with the live-time going beyond your project time-scale
- HEP experiments have a life-time of O(10) years:
  - It’s difficult to find a SW system that is likely to survive that long

Custom API for Metrics Monitoring

```
package Atlas.Monitoring;

interface Gauge {
    void setValue(double v);
}

interface Counter {
    void increment();
    void reset();
}

interface Metrics {
    Counter createCounter(String name)
                    throw (AlreadyExistsException);
    Gauge createGauge(String name)
                    throw (AlreadyExistsException);
}
```

*Makes it independent of the **Communication** implementation*

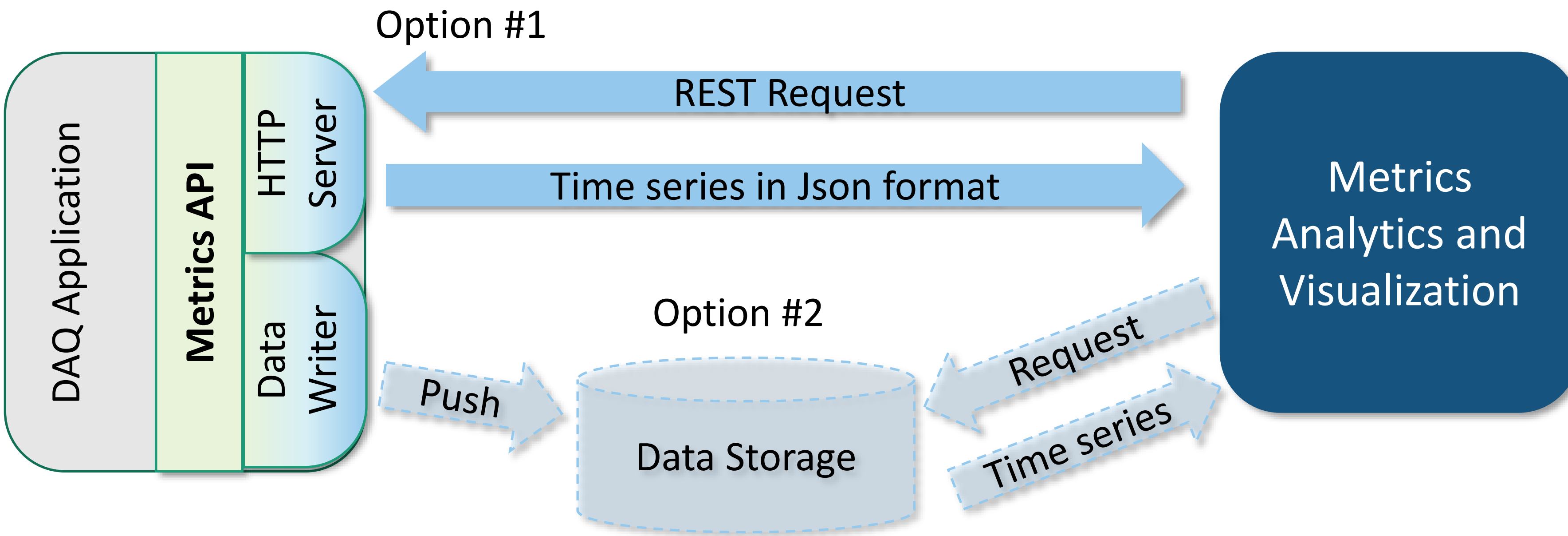
Supports different treatment for Counters and Gauges

Enforces uniqueness of Metrics IDs

Metrics IDs

- All Metrics must have unique IDs
- Uniform naming schema greatly simplifies Metrics handling:
  - Finding required Metrics is straightforward
  - Easy selection and filtering using regular expressions
- A possible approach:
  - System/Sub-system/Component/ + Metrics Name
- Examples:
  - */DAQ/DataFlow/EventRecoder/**EventsNumber***
  - */DAQ/DataFlow/EventRecoder/**RecordingRate***

Some Implementation Options



- The underlying implementation can be updated as the main project evolves:
  - Does not affect the DAQ applications
  - The same Analytics and Visualization tools can still be used

RESTful Protocol

- REST – **Re**presentational **S**tate **T**ransfer
- Client-server HTTP-based stateless communication protocol
- Supported by most of the modern information storage as well as Web-based Visualisation systems:
  - Supports seamless interoperations
- Makes it easy to switch from one Storage or Visualisation platform to another

REST Protocol Example

- Request:

```
https://atlasop.cern.ch/monitoring/
    ? id=ATLAS.Dataflow.RecordedEvents.Rate
    & from=now-30d
    & to=now
```

- Response:

Json Time Series, e.g.:


```
[
    {t:1579104640,v:12345},
    {t:1579104645,v:12346},
    {t:1579104650,v:12347},
    {t:1579104655,v:12348}
]
```

Web-Based Visualization Tools

- Javascript tools which work in Web Browsers:
    - **Grafana** - the open observability platform
    - **D3** – a low-level JavaScript toolbox for data visualization
    - **Rickshaw** – a JavaScript toolkit for creating interactive time series graphs
    - There are many others as well…
- Very convenient for the end users:
    - Don't require extra software installation
    - Provide real-time monitoring data access from any place of the World

Are there some other Advantages?

- The **API** can hide implementation of common data handling patterns

➢ Produce Derivative Metrics

➢ Perform Metrics Rate Down-sampling

➢ Keeps “Observer Effect” under control

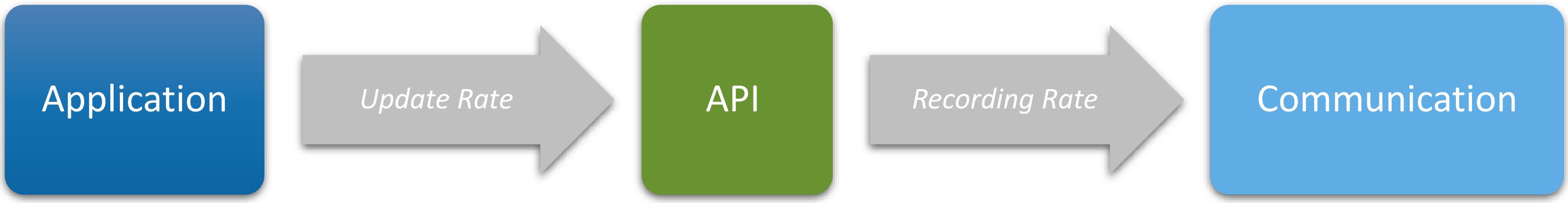
Derivative Metrics

```
import Atlas.Monitoring;

Counter events =
  Metrics.createCounter( "/DAQ/EventRecoder/Events");
…

void eventReceived() {
  evenets.increment();
}
```

1. "/DAQ/EventRecoder/Events"
2. "/DAQ/EventRecoder/EventsRate"

- Derivative Metrics can be automatically produced:
  - Counters => Rates
  - Gauges => Min, Mean, Max, Frequency distributions (histograms)

Metrics Rate Down-Sampling



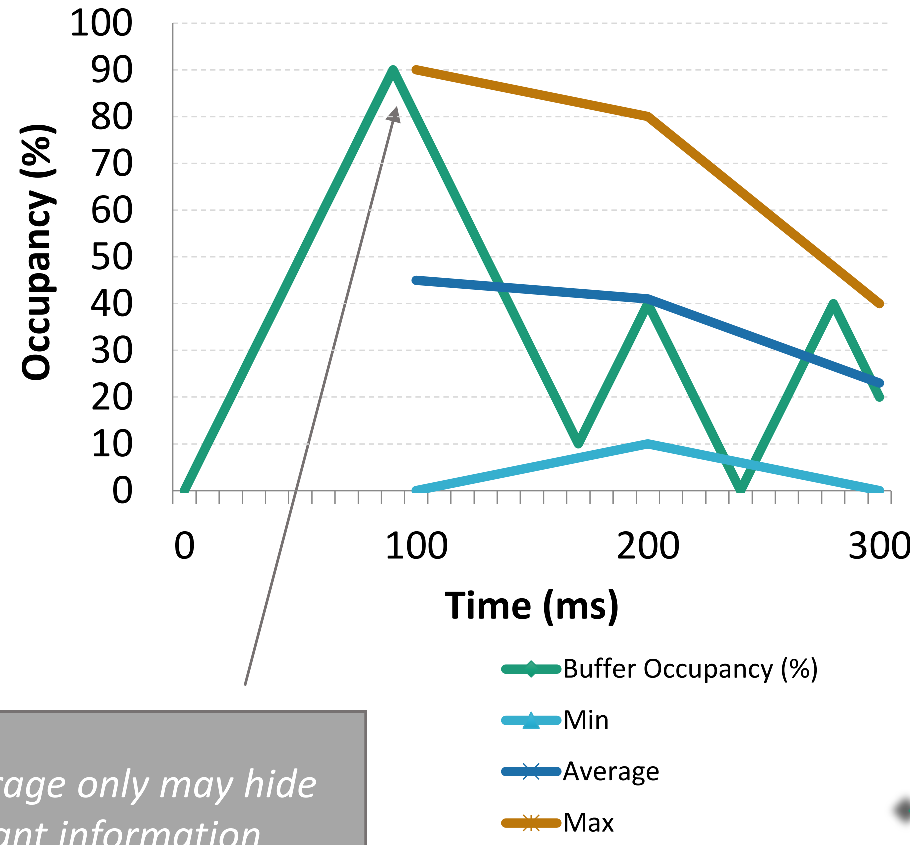
- Metrics update rate is defined by the data handling rate:
  - E.g. rate of triggers for the ATLAS experiment is 100 kHz
- High update rates must be scaled down:
  - Take too much space in the data storage
    - 100 kHz of event rate => $(8 + 8)*3600*10^5$ = ~6 GB data per hour per single metrics
  - Cannot be visualized:
    - 4K displays have 3840 pixels along X axis
    - Can display data for 40ms only

Metrics Rate Down-Sampling

- Metrics values can be down-sampled by the API implementation:
  - Reduces recording rate
  - Simplifies storage requirements
- Output update interval can be made configurable:
  - A default value for all metrics
  - Individual values per specific metrics
- Transparent for the **Applications** and **Communication** components

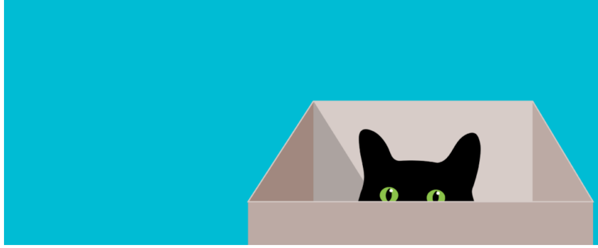
Down-Sampling: Counters vs Gauges

- Counter:
  - Publish the last value for each output update interval
- Gauge:
  - Publish three values for each output update interval:
    - Min, Average, Max



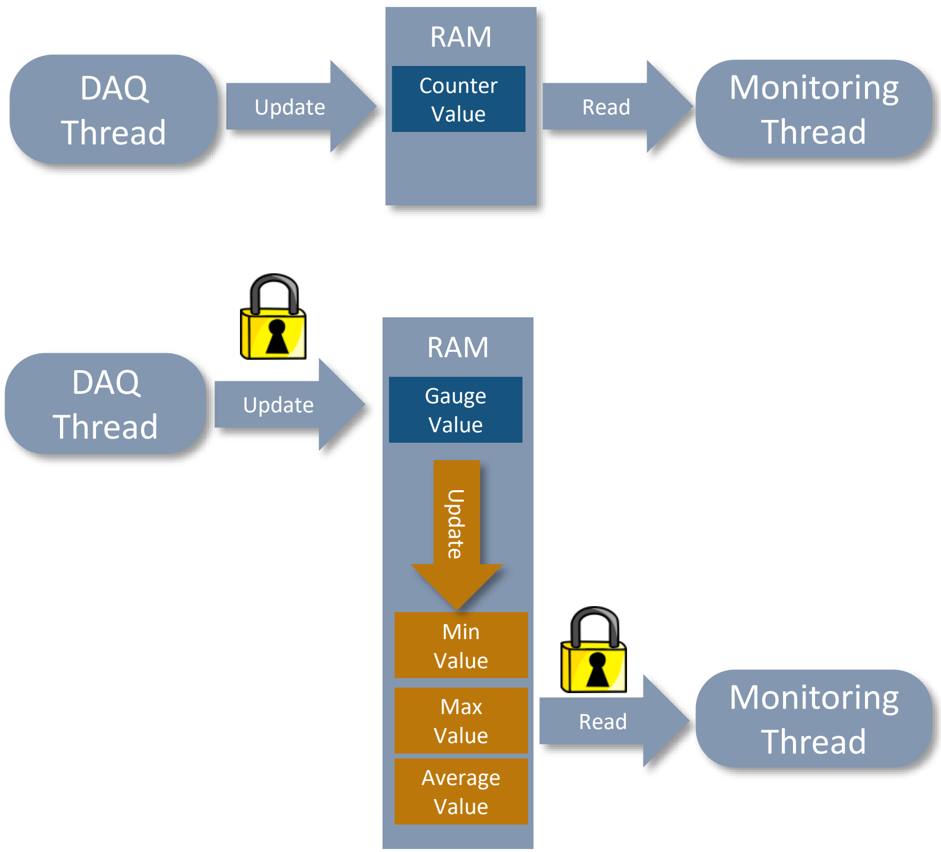
Using Average only may hide important information

The Observer Effect



- An observation affects the system:
  - It consumes resources (CPU, memory, network bandwidth)
  - It may affect performance of the monitored application
- Information must be passed to the **Communication** component **asynchronously**:
  - Monitoring information is updated by the DAQ thread
  - Down-sampling and publishing must be done by another thread
- Thread-safety must be considered:
  - But excessive thread-safety measures may hit the DAQ application performance

Thread-safety Overhead



- Counters don’t require a critical section:
  - Memory read/write operations on the modern Intel CPUs are atomic
- Gauge is different:
  - Monitoring Thread must not keep the lock when passing data to **Communication** component
  - Use a local copy

Thread-Safety Overhead

- Locking an unlocked mutex takes ~50 CPU cycles => less than 50ns:
  - If the mutex is locked this may lead to arbitrary delay
- Example: monitoring the buffer occupancy:
  - 10 kHz input rate:
    - Mutex locking takes 0.5ms every second => 0.05% overhead
  - 1 MHz input rate:
    - Mutex locking takes 50ms every second => 5% overhead

Scaling up the Monitoring System

The HEP Experimental Realm

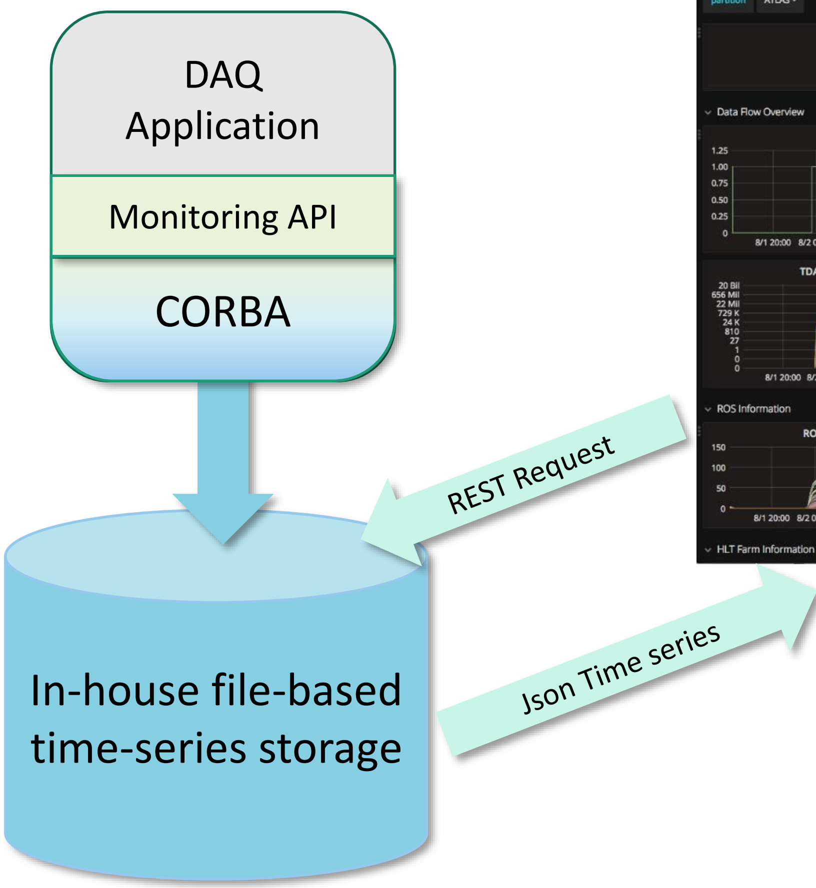
- A DAQ system of a modern HEP experiment consists of:
  - O(1K) computers and network devices
  - O(10K) SW applications
  - O(100K) Metrics

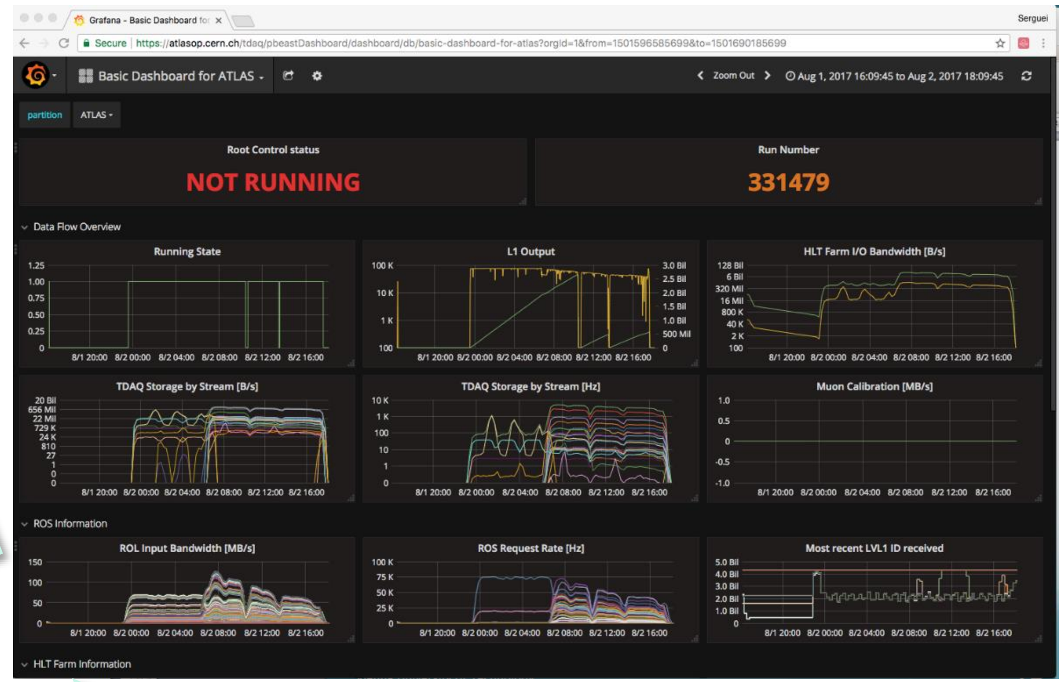
- A single gauge metrics for 24h run requires:
  - (8 + 8*3)*360*24=**280**kB of RAM
- 100K Metrics => 28GB per day => 200GB per week => **10TB per year**

Large Storage Implementations

- Traditional relational databases will not work well for large-scale projects
- NoSQL distributed alternatives:
  - **Whisper** – a lightweight, flat-file database format for storing time-series data
  - **InfluxDB** – a time-series database written in Go
  - **Cassandra** – scalable, high availability storage platform
  - **MongoDB** - a general purpose, document-based, distributed database

The ATLAS Experiment: Web-based Metrics Monitoring



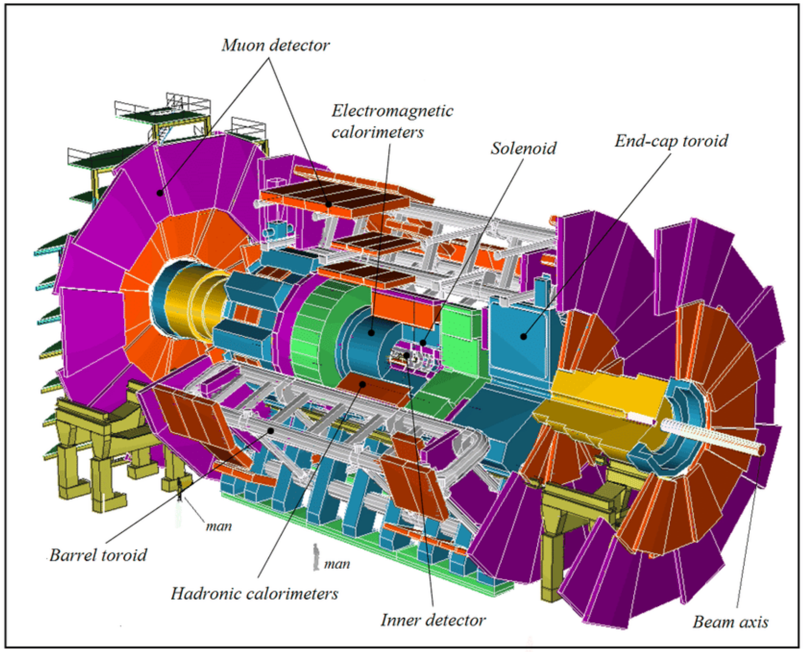


Grafana customizable dashboard

DAQ Specialty:
Data Quality Monitoring

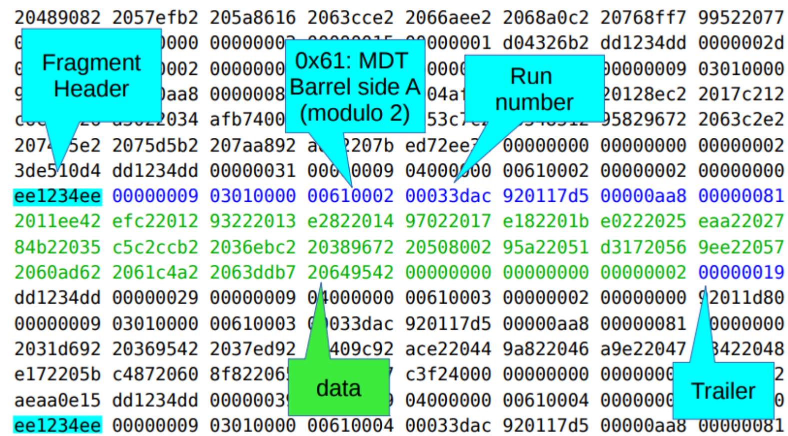
How to Monitor the Detector?

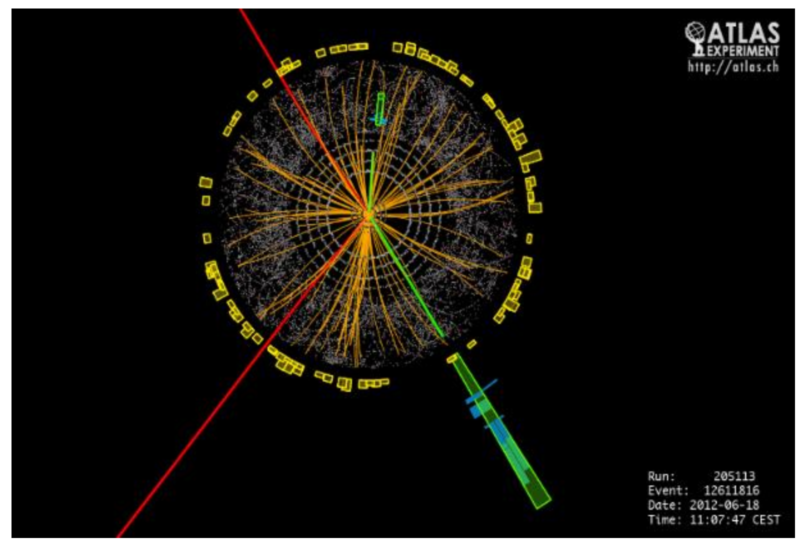
- Detectors of LHC experiments are incredibly complex devices:
  - Up to 10^8 output data channels
  - Mostly custom electronics
  - 40 MHz operational frequency
- Traditional monitoring would yield in O(1) PHz (petahertz) of metrics update rate:
  - These metrics are not even attempted to be produced
- However, DAQ system has a handle on these metrics…



Detector Metrics

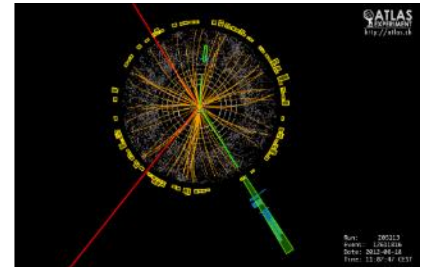
- Every **Physics Event** contains states of a sub-set of detector channels:
  - An expert can spot problems by looking into a graphical event representation
  - Such experts are not many and can't be in the Control Room 24/7





Automated Data Quality Analysis

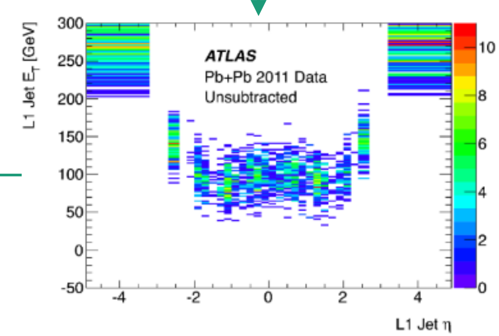
- Dedicated DAQ applications apply standard physics analysis algorithms to a statistical sub-set of ***Physics Events***:
  - Extract Detector Metrics and build their statistical distributions(histograms)
  - Analyze histograms and produce a new set of Metrics – Data Quality statuses



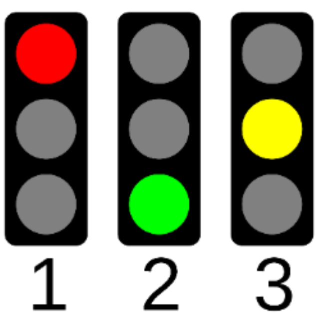
Samples of Physics Events

Physics Event Analysis Algorithms

Statistical Distributions



Statistical Analysis Algorithms



Summary: The Key Points

- ☑ Have your Monitoring System API ready from the beginning of the main project
- ☑ Use standard Monitoring APIs whenever it is possible:
  - e.g. Logging API
- ☑ Think carefully when designing a custom API:
  - It must not depend on a particular technology
- ☑ The Monitoring System implementation may evolve during DAQ system development
- ☑ Use existing solutions for Communication and Visualization components:
  - In-house development must be well justified