The CMS workload management system

M Cinquilli¹, D Evans², S Foulkes², D Hufnagel², M Mascheroni¹, M Norman³, Z Maxa⁴, A Melo⁵, S Metson⁶, H Riahi⁷, S Ryu², D Spiga¹, E Vaandering², S Wakefield⁸, R Wilkinson¹

¹ CERN, IT-ES Group, CH-1211, Geneva 23, Switzerland
² Fermi National Laboratory, Batavia, Illinois, USA
³ UC San Diego, La Jolla, CA, USA
⁴ California Institute of Technology, Passadena, CA, USA
⁵ Vanderbilt University, Nashville, TN, USA
⁶ University of Bristol, Bristol, UK
⁷ INFN Perugia, Perugia, Italy
⁸ Imperial College London, London, UK

E-mail: stuart.wakefield@imperial.ac.uk

Abstract. CMS has started the process of rolling out a new workload management system. This system is currently used for reprocessing and Monte Carlo production with tests under way using it for user analysis.

It was decided to combine, as much as possible, the production/processing, analysis and T0 codebases so as to reduce duplicated functionality and make best use of limited developer and testing resources.

This system now includes central request submission and management (Request Manager); a task queue for parcelling up and distributing work (WorkQueue) and agents which process requests by interfacing with disparate batch and storage resources (WMAgent).

1. Introduction

The CMS workload management system is responsible for data production and processing for CMS. A number of different types of computing and storage resources are available for these activities. Figure 1 shows the distribution of activities to resources. This model and its possible evolution is described in [1]. The resources available to CMS are shown in Table 1.

These resources are provided by 1 Tier-0, 7 Tier-1 and ~50 Tier-2 sites. O(100,000) jobs are required to be running continuously on these resources to carry out CMS’s various computing activities. These jobs run the CMS software framework (CMSSW) which is capable of reading, writing and processing CMS data. This software is driven by a configuration file that describes the inputs, outputs and processing to be carried out. It is the job of the workload management system to provide this. The system needs to manage and automate the process, e.g. identifying new data to run over, cope with job failures etc. The output from the jobs is then registered with the data management (DBS) [2] and transfer systems (PhEDEx) [3]. This paper will discuss the workload management system and how this is managed.

Previously the workload management system was composed of 3 separate code bases:

- The production and processing system, and
- The Tier 0, and
Table 1. CMS computing resources.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Tape (PB)</th>
<th>Disk (PB)</th>
<th>CPU (number of cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERN</td>
<td>24</td>
<td>5</td>
<td>~15,000</td>
</tr>
<tr>
<td>Tier 1</td>
<td>50</td>
<td>22</td>
<td>~20,000</td>
</tr>
<tr>
<td>Tier 2</td>
<td>-</td>
<td>26</td>
<td>~45,000</td>
</tr>
</tbody>
</table>

- The analysis system.

The production and processing system was based on software called ProdAgent [4] illustrated in Figure 2. This had been designed for producing simulation data and had been adapted to also provide data processing functionality. The system therefore had a number of flaws when it came to processing which led to a small but significant failure rate. There were many ProdAgent’s with each usually responsible for a given activity and/or group of resources. Instances were not linked and required a command to be run manually on the machine hosting the ProdAgent to feed it work. Splitting work between multiple instances required care to ensure that there were no overlaps.

![Figure 1. The CMS computing model (inter-tier data-flows not shown).](image)

An earlier consolidation effort had allowed the Tier 0 system to use the prodAgent for low level functionality [5]. This allowed the developers to concentrate on the highly specialized behavior required for this use case. However it suffered from the same problems as the processing system.

At the same time the analysis software, which fulfilled much the same role as the processing system, was largely separate [6]. Thus functionality was duplicated between these two systems and problems encountered by one were encountered by the other without the benefit of shared knowledge. The analysis system could optionally use a ProdAgent for job submission and tracking however the majority of functionality (job creation, job runtime code) was custom and duplicated similar code in the ProdAgent system.

Experience with these systems led to an increased familiarity with the problem and it was decided to develop a new system designed to solve these limitations. It was decided that the
new system should act as a common layer that provided workload management solutions for all CMS software. This system would consist of: a library layer which provided general workload management functions and a higher layer where these low level functions could be tied together to provide the functionality that the CMS workload management applications required.

CMS had avoided a central system managing all of its distributed computing workflows, believing that independent instances (or agents) allowed for greater scalability and reliability. This approach did have a number of disadvantages though: only the Tier-0 had an automated way for work to be handed to the system, and there was no central record of what had been run in either the production, processing, or analysis systems. Thus it was decided that the benefits of a distributed, scalable system would be kept by having separate agents (termed WMAgents in the new system) submitting and managing jobs but that joining these together with central work acceptance and allocation systems would provide large benefits. Work would be entered and logged in the Request Manager (ReqMgr) and distributed to WMAgents by the WorkQueue [7]. This new system is illustrated in Figure 3.

In addition to this another system was written to help manage the creation of CMSSW configurations and their management [8]. The configurations passed to the ReqMgr come from this system.

2. The Request Manager (ReqMgr)

The problem with traceability and provenance was solved by the addition of the Request Manager. This is a website which takes the specification for the work to be run, including input/output data and CMSSW configuration. This is called a request and is given a unique name which allows the requestor to follow its progress. The Request Manager takes the information passed in by the requestor and creates a workflow. This is a Python [9] object which holds all the information needed to be given to the WMAgent to create and manage the jobs necessary to carry out the specified work. ReqMgr is composed of:

- A web server which includes a REST [10] interface.
- A CouchDB [11] NoSQL database which is used to store workflows and CMSSW configs.
- A SQL database which holds the data not stored in CouchDB, i.e. known users, groups, CMSSW software versions, and other parameters which require validation against known good values.

![Figure 2. Typical ProdAgent workflow.](image-url)
Figure 3. New workload management architecture.

ReqMgr (Request Manager) is the heart of the Workflow management system which controls the state machines of the Workflows: assigned, running, completed and closeout.
When a request is submitted the input parameters are validated. This catches a lot of errors that previously would only have been detected once jobs failed, or worse when it was noticed their output was corrupt.

The ReqMgr allows a requestor to specify a priority when submitting a request. This is combined with a priority associated with the group to which the requestor belongs. This allows the system to manage relative priorities between communities.

3. The WorkQueue

To improve reliability and automation in the system the WorkQueue [7] was introduced which takes requests from ReqMgr, assesses priorities and feeds them to the agent best placed to carry out the work. The WorkQueue functions as a central task queue which allows the agents to be considered as transient, allowing instances to be removed or added as needed with no break in activity.

The WorkQueue does not contain individual jobs. This would require the WorkQueue to run the computationally heavy algorithms that are better suited to run in the WMAgents. Instead the WorkQueue splits the request into chunks (or elements) which represent a reasonable amount of work.

The WorkQueue system consists of a global WorkQueue which obtains requests from the Request Manager and a local WorkQueue which is part of the WMAgent. The local queue pulls work from the global queue and allows for work to be buffered close to the agent, reducing the latency for acquiring work and mitigating periods when the global server is unavailable.

The WorkQueue consists of two databases; the Inbox and the main element database. In the global WorkQueue the element database contains the elements that the local queue works on. In the local queue the Inbox is kept synchronized with the ReqMgr, in the local queue it is kept synchronized with the global WorkQueue element database.

A workflow controls how the WorkQueue expands it into elements by setting a splitting policy (and parameters), see Figure 4. This allows the WorkQueue to adapt to workflows which vary widely. Currently the range of policies is quite limited. Workflows requiring input data are split based on groupings (blocks) of input files. In CMS’s data management system a block of files are moved around together which allows a reduction in the number of queries needed of the file location service. Workflows not requiring input data are split into evenly sized blocks based on the estimated amount of computation required. More advanced policies are possible, such as only allowing a subset of a workflow to be run initially to act as a validation step which can catch problem workflows before significant resources have been wasted.

![Figure 4. WorkQueue start and end policies.](image-url)
Another policy is used to aggregate the status of the elements in order to report the workflow status back to the ReqMgr. More advanced features have yet to be implemented but could include: the release/cancellation of the workflow based on the validation sample or using failure statistics for the whole workflow as a trigger for stopping processing and reducing any waste of compute resources.

A workflow contains a priority which the WorkQueue uses to schedule elements against each other. To prevent low priority work starving in the queue while high priority obtains resources, the WorkQueue increases an element’s relative priority based on the length of time the element has been queued.

The WorkQueue is based on Apache CouchDB, a document oriented NoSQL database. The WorkQueue uses many CouchDB features (map/reduce views and bi-directional replication between distributed instances) in its operation. CouchDB provides both database and web server functionality. The widely varying workflows the WorkQueue has to work with fit well with CouchDB’s schema-less document oriented feature set. One of CouchDB’s key features is the ability to pre-define queries on the stored data which can use high level languages such as JavaScript; these are known as views. The results from a view are stored and only recomputed when an underlying element changes, this allows them to be performant.

CouchDB includes a replication feature which is used to keep the global and local WorkQueues’ synchronized. When an element is acquired by a local queue it is marked as belonging to that queue. The CouchDB in the local queue is then configured to keep elements synchronized between the two instances.

When a workflow has finished processing in the agent the local WorkQueue runs the relevant end policy to obtain an overall status. The elements in the database are then deleted. Once the status is propagated to the global WorkQueue, the global WorkQueue runs the relevant end policy and updates its status. The elements in the global WorkQueue are then deleted; this deletion is replicated to the local WorkQueue which removes the remaining knowledge of that workflow from its database. The ReqMgr is updated with the final status and then the global WorkQueue deletes all records of the workflow.

4. The WMAgent
The WMAgents provide the bulk of the workload management system functionality. From a workflow they create, submit and track the jobs that carry out the desired work. The WMAgent consists of three main parts, shown in Figure 5.

- Local WorkQueue - interface to the global WorkQueue.
- Workload Management Bookkeeping System (WMBS) - describes the workflow to data mapping.
- Job State Machine (JSM) - creates and manages jobs.

The local WorkQueue was described in detail above. It obtains and queues work from the global WorkQueue. When it identifies free resources it runs the same code as the global WorkQueue to match these to WorkQueue elements. These are then injected into the agent via the WMBS database. After work has been injected the WorkQueue pulls more work from the global WorkQueue and buffers that so that it has work ready for when the agent has more free resources.

The WMBS contains the mapping between workflows and data, this is known as a subscription. In the case of a processing workflow, when a WorkQueue element is injected a mapping is made between the workflow and the input files to be run over. For a simulation workflow the WorkQueue creates a fake file which contains the information to direct the job creation.
The JSM records the state changes of each job in a CouchDB database and ensures that only allowed transitions are made. The JobCreator, a JSM component, reads the information in the WMBS database and runs the appropriate job splitting algorithm to convert the information from WMBS into jobs.

The JobSubmitter takes the jobs to be submitted and submits them to the available resources. It takes the input file locations for a job and matches them against the resources known to the agent. Sites are defined to an agent as a list of resources. This abstraction allows for the underlying sites to be organized in multiple ways. For instance, different multiple sites can be represented as one abstract resource. This provides great flexibility. Each resource defines entry points for compute and storage resources along with the submission protocol the compute endpoint understands. The storage endpoint is simply the domain name of the storage and does not specify how to contact it. This is because all access to the storage is performed from jobs sent to the site and once there they obtain site specific configuration which specifies how to access the storage.

It also takes account of directives in the workflow which can specify particular sites to use or to avoid. The JobSubmitter (and tracking) system is designed to use a plugin system for communicating with the actual job submission technologies. This allows each resource to be used with the most appropriate technology and a single agent to manage jobs on disparate systems. This plugin system is known as BossAir, and is a lightweight implementation of a previous job submission and tracking system. When jobs are submitted they are stored to the BossAir database which is used to track their progress as they run.

BossAir supports both the gLite WMS [12] and pilot jobs via the GlideIn WMS [13], as well as direct submission systems, e.g. LSF, Condor etc. CMS has used both the WMS and GlideIn’s for a number of years. However the pilot based system, as provided by the GlideIns, allows for better error handling and for scheduling decisions to be taken as late as possible. It has thus been decided to deprecate usage of the gLite WMS; it is expected that usage will decline over the next year or so until the GlideIns are the only supported grid scheduler.

Once submitted the jobs are monitored by the JobStatusLite component which watches the jobs for state changes, i.e. when they start running. This state is stored in the BossAir database for use by other components. Due to experience with the gLite WMS it also detects jobs which are not progressing through states at a reasonable rate. This generally indicates some problem with the gLite WMS, which has a known failure mode where jobs can remain stuck in one state. Increased adoption of the GlideIn WMS should reduce the occurrence of this problem. If a job gets stuck, the component attempts to kill it in the WMS and declares it a failure.
Failed jobs are detected and handled by the ErrorHandler. This detects jobs which have failed either during creation (as detected by JobCreator), submission (detected by the JobSubmitter) or after submission (detected by JobStatusLite). The general response to a failed job is to retry the failed action. For the majority of intermittent failures, e.g. storage access problems, this is the correct recovery procedure. In the case of some intermittent problems retrying immediately may not be the best action, e.g. a site may require some time to fix the problem. Because of this the ErrorHandler does not trigger an immediate retry but instead moves the job into an intermediate cooloff state, where it waits a suitable interval before the desired action is retried.

Persistent problems will not be solved by retries so after a fixed number of attempts the job will be placed in a failure (exhausted) state. Some permanent failures can be identified by the CMSSW framework, these are indicated by the error message and exit code produced. Generally these indicate a fatal configuration error. The ErrorHandler is capable of detecting these and skipping the retry phase. This provides quicker feedback of misconfigured workflows.

Once jobs are in a cooloff state the RetryManager is responsible for deciding when to release them. This is based on the failure type and the algorithm the component is configured with. Initially an algorithm which used a fixed cooloff time was used. However this was found to retry too aggressively and often caused jobs to be failed before intermittent problems could be solved. Hence the new default algorithm backs off after each failure. This allows a combination of quick initial retry attempts, which can solve some intermittent problems, while allowing a longer period for more serious problems to be fixed. Failing a job after a number of failures is not always acceptable, i.e. in the Tier-0 where all effort needs to be made to complete jobs successfully. In this case the RetryManager has an algorithm which will pause jobs. The job will stay in that state until it is manually moved to a retry state, if the problem has been solved, or to a fail state, if the problem cannot be fixed.

The JobTracker monitors BossAir for finished jobs (either successfully or not) and marks them in the JSM as finished (or failed). Successful jobs are then processed by the JobAccountant. When a CMSSW job finishes it produces an XML file which describes the inputs, outputs, processing carried out and useful metrics, e.g. memory usage and IO statistics. This report is returned with the job and is used by the JobAccountant. The file input and output information is saved to WMBS. This is needed because most CMS workflows contain multiple steps where subsequent steps operate on the results from previous steps. The most common case is for a first step to produce output files which are then merged together by another step to form large (2GB) files. To facilitate this the output from the first step is set as the input for a merge subscription. Once input data is available the JobCreator runs and creates the necessary merge jobs which go through the same submission and tracking process as the previous jobs.

The mapping from input to output files must be saved to the central CMS metadata catalog. The JobAccountant saves the file and dataset information to a local database, the DBSBuffer, which is uploaded to the central metadata catalog [2] in a bulk operation. Once files have been uploaded by the DBSBuffer to the global database another component, the PhEDExInjector, uploads the file locations to CMS’s transfer system.

When the necessary information has been uploaded the job can be removed from the agent, this is the job of the JobArchiver. When all jobs in the workflow have finished the output dataset is marked as closed in WMBS, this informs the rest of the system that no more data will be added and allows the dataset finalization actions to run. This includes taking the metrics from the individual jobs and summarizing them for the workflow. This information is uploaded to a central repository where it is available for later study.

When the subscriptions in a workflow have finished, the TaskArchiver deletes the information used to create the summary from the agent. The WorkQueue is then notified about the finished subscriptions and marks the appropriate WorkQueue elements as complete. This is then propagated up to the ReqMgr and the workflow deleted from both the global and local
5. Tier-0
The Tier-0 is currently running on the old workload management system. It has custom data discovery and job creation but uses a ProdAgent for job execution. This system has been in use since 2008 and a number of architecture decisions taken in the Tier-0 design were later adopted in the WMAgent. The Tier-0 is being rewritten to be based on a WMAgent and to fully take advantage of the new features provided. For instance the Tier-0 will take advantage of the WMBS database to drive job creation where as previously it had its own custom data to job mapping. More details of this activity can be found in [5].

6. Analysis
Allowing the distributed analysis system to take greater advantage of commonalities with other CMS workload management systems was one of the prime drivers for the new system. This activity is almost complete and is due to begin intensive integration testing soon. The current system optionally utilizes a ProdAgent, from the old workload management system. The same problems which affected organized data processing also affected the analysis use case, however the impact of these problems was greater due to the average analysis user lacking the expert knowledge necessary to fix these issues. The new analysis architecture is shown in Figure 6. The majority of components are the same as in the production system however different instances will be used to avoid one activity impacting another. The only custom components will provide analysis specific functionality, such as holding user configuration files and libraries.

One of the main problems with the current analysis system is the treatment of output files from a job. Each user is allocated space on a Tier-2 for the output of their analysis jobs however these files are copied from wherever in the world their jobs run. This can involve significant transfers across continents which is both slow and prone to failure. This is responsible for approximately half of all analysis job failures [14]. In the future analysis jobs will adopt the strategy used in the organized activities which is to save output to the storage local to the job
and then have another process copy it to the desired end location. For analysis jobs this function will be provided by the new AsyncStageOut component. More details on this component are available in [14].

The analysis system is under heavy use and to ensure physics discoveries can still be made the new system needs to be tested thoroughly before adoption. Thus the proposed system has been scale tested to over 4 times the current level of 250,000 jobs per day. For more details see [15].

7. Usage
This system has been gradually rolled out over the last two years. By 2012 the majority of organized CMS activity (with the exception of the Tier-0) was using it. In Figure 7 this is the production and reprocessing categories. As mentioned above the analysis system is still using the old system but is in the process of testing before beginning the switch. The Tier-0 is in a similar situation, though it is also constrained by the CMS data taking schedule for any changeover.

8. Conclusion
The CMS workload management system has been largely rewritten to take advantage of lessons learnt in the early years of CMS data taking and to allow for greater consolidation between related workload management applications. A significant fraction of CMS’s activities have been ported to this system and are seeing major benefits. More details on usage can be found in [16]. The Tier-0 and analysis activities are both in the process of moving to the new system. The new system will also allow CMS to take advantage of new relevant technologies such as multi-core jobs that may become relevant in the future.
References


