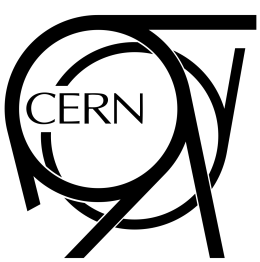
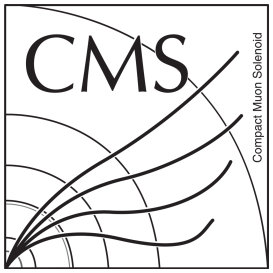
CMS: Modernizing a CMS Production Code Base





Shivam Singhal

singhals.3316@gmail.com

Supervised by

Hasan Öztürk

Muhammad Hassan Ahmed

A project report submitted in partial fulfillment of the requirements for the CERN Summer Student Program 2024

Summer Student at CMS O&C group

CMS

CERN

Geneva, Switzerland

August 2, 2024

1 Abstract

Unified is a tool employed by the CMS offline and computing team to preform checks during the scheduling process of physics workflows on the computing grid available to the CMS group. The whole framework comprising of the WMCore and Unified operates as a finite state machine, starting from requesting a workflow to be scheduled on one of the available nodes. The process includes verifying the credentials of the requester, performing various validity checks on the workflow, scheduling and executing it, and assessing whether the produced results meet a minimum threshold. If the results meet the threshold, the workflow is announced; if not, it is resubmitted as an ACDC workflow.

Despite its critical role, Unified was originally written without adhering to coding best practices. It featured several scripts/services comprising of a single, large function handling all tasks, resulting in redundant code and the absence of dedicated classes where necessary. This led to a complex and difficult-to-understand codebase. The task of this project was to refactor these scripts to improve their structure and maintainability.

2 Introduction

Unified was originally created to automate the manual steps taken for day-to-day P&R operations. It is a set of scripts that have critical roles in production that run as croncobs. It was originally named **WmAgentScripts** as it was a set of scripts to be run on the WMAgent machines to automate the manual steps taken for day-to-day P&R operations. It was later renamed to **Unified**. Figure 1 shows the evolution of the WmAgentScripts/Unified repository ,through the major transitions it has gone through.

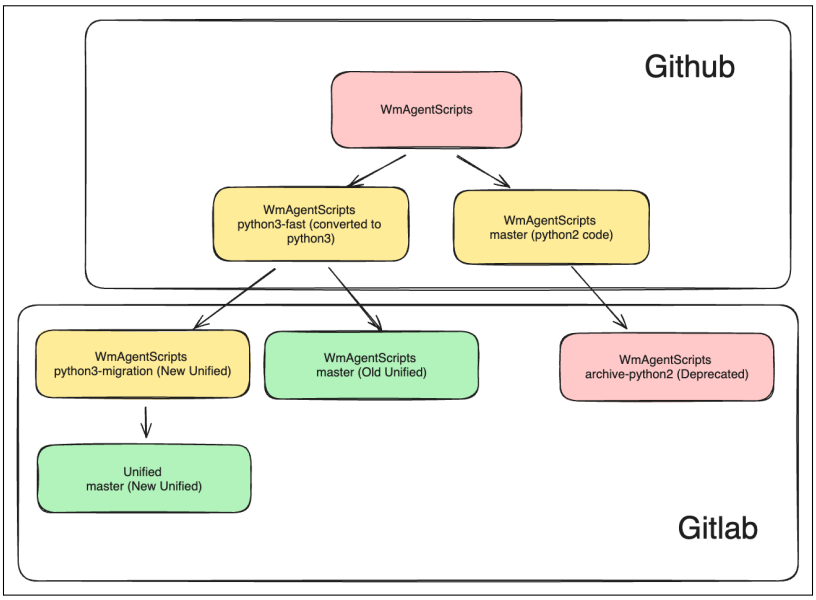


Figure 1: *The Green box represents the current active branches, the red box represents the deprecated branches and the yellow box represents the transitional branches created during the process.*

Unified was originally written in Python2 and was hosted on GitHub. It was later migrated to GitLab, and the Python3 migration was started in 2021-22. With the Python3 migration, there were two major versions: **python3-migration** and **python3-fast**.

The `python3-fast` was just the Python2 code converted using tools to work in Python3. The `python3-migration` was the ground up rewrite of Unified to optimize the code and make it

more efficient. The `python3-fast` was renamed to **master**, and `python3-migration` moved to a separate repository **New Unified**.

The Python2 version was archived in a separate branch, **archive-python2**.

`Unified Modules` are the scripts that run as cronjobs. They are responsible for various tasks in the Unified system. Figure 2 shows the life cycle of a workflow as it is submitted to be scheduled through all the possible states. Several of these transitions require checks performed by `Unified Modules`

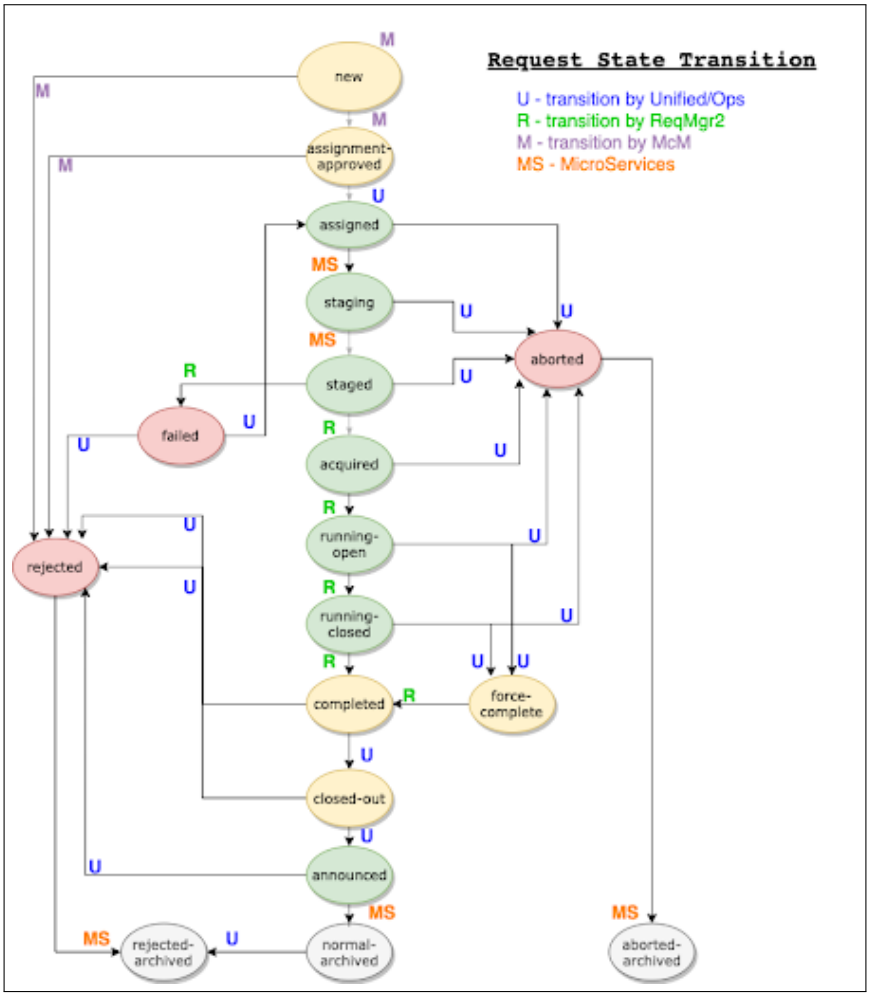


Figure 2: *Life Cycle of a workflow in production*

Apart from these scripts, several of the state transitions are performed by `ReqMgr`(a service owned by WmCore) and various other microservices. These services, together with the `Unified Modules`, create a framework being used by the CMS O&C group to run workflows on their computing grid.

The current scripts suffer from several maintenance issues due to suboptimal coding practices. These deficiencies lead to significant challenges such as:

1. **Lack of Maintainability:** Once the original developer exits the project, the codebase often remains unmaintained, resulting in numerous obsolete and malfunctioning scripts.
2. **Onboarding Challenges:** New developers face difficulties in comprehending the code's functionality and architecture due to insufficient documentation, lack of modularity, and poor code readability.

3. **Technical Debt:** The accumulation of technical debt due to quick fixes and inadequate refactoring efforts exacerbates long-term maintenance and scalability issues.
4. **Inadequate Testing:** Lack of comprehensive unit tests and integration tests results in fragile code that frequently breaks when changes are introduced.

To understand the working of Unified and to tackle the above mentioned challenges, the migration of `Old Unified` to `New Unified` was started. The aim of this summer project was to contribute to the ongoing work and speed up the migration process. As a summer student, I contributed by refactoring a major script called the **Assignor** alongside some minor scripts such as **ConfigurationHandler** and **Resubmit**. The functioning of these scripts have been talked about in detail in the following sections.

3 Assignor

Assignor, in short is a script which fetches all the workflows from `ReqMgr` having status as `assignment-approved`, run some checks on it, assign some properties to it, commit back to the `ReqMgr` the updated metadata for the workflow and sets its status to `assigned`.

assignment-approved

- Requests in this status are awaiting review and assignment from the Unified/Campaign Manager. They will be moved to rejected if there is a problem with the request, otherwise it is assigned. (McM, Physics groups)

assigned

- These requests have been provided with an appropiate site whitelist, acquisition era, processed dataset and other attributes. Requests in this state are examined by the `WorkQueue` and moved to `acquired` after creating the work elements, or in case of failure the request is moved to `failed`. (Unified, Ops)
- This is also where Unified takes a look and decides if the workflow should be run as a TaskChain or a StepChain. If it would be more efficient to run as StepChain, the workflow is rejected and cloned as a StepChain Workflow.

The assignor is the script which sets the status of a workflow from `assignment-approved` to `assigned` and is therefore responsible for attaching the site whitelist, acquisition era and other properties after performing required checks.

3.1 AssignmentProcessor

`AssignmentProcessor` is a class in the new Unified script comprising of all the business logic of the assignor. It includes the functions to check the various properties of a workflow and also to assign some other properties to this workflow, such as the dataset on which it has to run required further while running the workflow. A brief description of each of its method has been provided in Section 10.1

3.2 Assignor

`Assignor` is a class in the new Unified script comprising of all the functions related to fetching the workflows, calling the functions to assert checks and assign other properties and finally committing the workflows to Unified and ReqMgr. A brief description of each of its method has been provided below:

1. `go:` Determines if the injector can proceed by checking for module locks and verifying the status of certain services. Returns True if the injector can proceed, and False otherwise.
2. `commitWorkflowToUnified:` Commits the current session to the unified database if the `'force_assign'` option is not set.
3. `getPendingWorkflowCount:` Retrieves the count of workflows in specific statuses ('assigned', 'staging', 'staged', 'acquired') from the request manager and returns the total count.

4. `getSortedWorkflows:` Retrieves workflows for assignment, sorting them by priority if not `'force-assigning'` by making calls to `_getUnifiedWorkflows` or `_getUnifiedWorkflows`.
5. `initialChecks:` Performs a series of initial validation checks on the workflows, returning an appropriate code to indicate whether to proceed, break the loop, or skip the workflow.
6. `modifyWorkflowParameters:` Modifies various workflow parameters by invoking several other functions for parameter updates, validation, and configuration.
7. `assingWorkflow:` Handles the assignment of a workflow by using a WorkflowAssignmentProcessor to process the workflow and then checking the result of the assignment.
8. `assignWorkflows:` Manages the assignment process for a list of workflows, performing initial checks and parameter modifications before assigning each workflow.
9. `run:` Initiates the assignment process by fetching and sorting workflows, and then assigning them.

The definition of the remaining functions can be found in Section 10.2

4 Resubmit

The `resubmit.py` script is designed to handle the resubmission and modification of workflows within a workflow management system. It provides two primary functionalities: cloning and extending workflows. Below is a detailed breakdown of its operations.

4.1 Workflow Cloning

- **Class:** `WorkflowCloner`
- **Purpose**: To create a clone of an existing workflow with potentially modified parameters.
- **Key Methods**: `cloneWorkflow`: Main method that retrieves the workflow schema, modifies it, sets necessary parameters, submits the workflow, and approves it.

4.2 Workflow Extending

- **Class:** `WorkflowExtender`
- **Purpose**: To extend an existing workflow by adding more events or modifying its parameters.
- **Key Method**: `extendWorkflow`: Main method that retrieves the workflow schema, modifies it, submits the workflow, and approves it.

4.3 Execution Flow

- The script starts by parsing command-line options and user inputs using `WorkflowResubmitManager.parseOptions()`.
- Based on the specified action (`clone` or `extend`), it iterates through the list of workflows and performs the respective operation using `WorkflowCloner` or `WorkflowExtender`.

5 New Features and Bugs

5.1 Feature: ConfigurationHandlers and testing environments

To configure whether a script runs in production or testbed, a new approach for creating singleton objects of various configuration handlers was proposed. For future `Unified` scripts, the workflow should be as follows:

1. At the very beginning, determine if the script is to run in production or testbed by parsing the command line arguments.
2. If running in testbed, create an instance of the `ServiceConfigurationHandler` class with the parameter `testbed=True`. This will configure the `ServiceConfigurationHandler` to read from `testServiceConfiguration.json`, which contains the URLs for all testbed services.

3. Subsequently, when invoking any other configuration handler class, only a single instance will be created, connected to either production or testbed. This connection is determined by whether your `ServiceConfigurationHandler` instance reads from `testServiceConfiguration.json` or `serviceConfiguration.json`, as configured in previous step.

This is a more efficient approach as we do not have to create several objects to read the same configuration file to be used by different classes. Also the singleton object is easily configurable to connect to the production or testbed services as described in the above approach and therefore we only have to initialise at one place. This also saves us from mistake of specifying the configuration handlers for testbed environment separately each time.

Besides this a testbed mongoDb server wad created on `vocms273.cern.ch` and only accepts requests from `vocms273.cern.ch`. Therefore, it is recommended to always run the scripts for testbed on `vocms273.cern.ch`

5.2 Bug: ModulelockController instantiation

The following function is used to check whether all the required services are reachable and the script runs further only when this function returns true. There is a small bug in this function in line 7. The `moduleLockController` object has to live for the entire execution of the script but as the object is local, it gets destroyed as soon as the function returns and thus the rest of the script is running without holding a lock on the Mongo service.

```
1  def go(self) -> bool:
2      """
3      The function to check if the injector can go
4      :return: True if it can go, False o/w
5      """
6      try:
7          moduleLockController = ModuleLockController()
8          servicesChecker = ServicesChecker(softServices=["mcm", "wtc",
       "jira"])
9
10         return (self.options.get("manual") or not moduleLockController
       .isLocked()) and servicesChecker.check()
11
12     except Exception as error:
13         self.logger.error("Failed to check if Injector can go")
14         self.logger.error(str(error))
```

To workaround this bug, a small Fix is suggested on `line 7`:

```
1  self.moduleLockController = ModuleLockController()
```

With this modification, the `moduleLockController` object will live as long as the object of the class(in our case `Assignor`) lives and therefore it will hold lock for the entire execution of the script. But remember to call the function while initialising the class (i.e. from the `__init__` method of the respective class) otherwise, the object may never get created.

The issue can be accessed from here: ModuleLockController not holding lock during entire execution of the script

6 Final Run and Results on the testbed

A complete integration testing pipeline was assembled to test the execution of the both the `Assignor.py` and `Resubmit.py` script. The execution flow of the script was:

1. **parseArguments**:
   - Parses command-line arguments and returns them as a dictionary.
2. **Main()**:

- Parses arguments before running the unittests by calling the function `parseArguments()`
- Uses `unittest.main()` and the `parsed_args` are passed as global variable to the script

3. **AssignorIntegrationTest Class**:
   - `setUpClass()`: Initializes class-level variables using `parsed_args` and sets up the service configuration handler, Oracle client, ReqMgr writer, and testbed status.
   - `run_script_with_file()`: Runs the `Resubmit` script with a file input, capturing and returning output.
   - `run_script_with_workflows()`: Runs the `Resubmit` script with specific workflows, capturing and returning output.
   - `process_outputs()`: Processes script outputs to extract new workflow names.
   - `update_oracle_db()`: Adds new workflows to the Oracle database and verifies their status as `'staged'`
   - `assign_workflows()`: Assigns workflows using the Assignor script.
   - `check_statuses()`: Checks and verifies the status of the new workflows in both the Oracle database and the ReqMgr as `'away'` and `'assigned'` respectively.
   - `abort_and_reject_workflows()`: Aborts and rejects new workflows, updating their status in both the Oracle database and the ReqMgr.
   - **test_process_workflows()**: Main test method that:
     a. Retrieves workflows or file paths from arguments.
     b. Runs the `Resubmit.py` script based on the input.
     c. Processes outputs to get new workflows.
     d. Updates the Oracle database with new workflows.
     e. Assigns workflows using `Assignor.py` and checks their statuses.
     f. Aborts and rejects workflows.

The Figure 3 shows the state transition of a workflow using the integration testing script, it first *clones* a workflow, sets its status to *assignment-aaproved*, runs the *assignor* script on it and finally after successfully assigning it *aborts* the workflow.

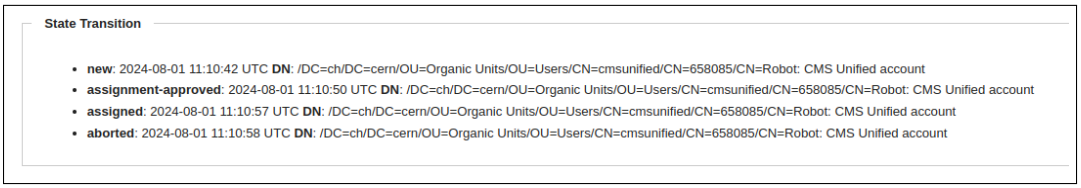


Figure 3: *State Transition of a workflow*

7 Conclusion and Future Work

The future plan for P&R is to migrate the entire codebase of the `Unified` framework to be managed by `WMCore` services. Because of this, the need to thoroughly understand the existing code and modify it to seamlessly integrate with the interfaces provided by WMCore arises. The goal of this project was thus to make the code more modular, reducing redundancy and enhancing readability, maintainability, and ease of future modifications.

One significant task achieved through this project is the migration of the `assignor` script. Additionally, a major script that still needs to be migrated is the `checkor`. The `checkor` script is responsible for monitoring workflows in various statuses, managing transitions from `completed` to `closed-out`, handling assistance labeling, performing duplicate checks in output datasets, and conducting several other essential checks.

8 Acknowledgement

Dedicating the summer to this project was a great experience, made possible by the support of several people. Firstly, I would like to thank my supervisors, Muhammad Hassan Ahmad and Hasan Öztürk, for kindly answering all my questions and helping me improve my coding skills. I cannot thank them enough for allowing me to have this unique opportunity.

I am also thankful to the CERN Summer Programme Team for all their hard work in making this possible, and to all the CERN members for all the knowledge that these great minds working here shared which have ignited a fire within me to do more.

I certainly look forward to continuing my work at CERN and embracing new challenges in the future.

9 References

1. Processing and Production The Lifecycle of a Workflow O&C Week
2. CMS PnR Documentation

10 Appendix

10.1 AssignmentProcessor

A brief description of each of the functions implemented in `AssignmentProcessor` can be found below:

1. `checkSpecific`: Verifies if the current workflow same as the specified workflow, allowing assignment if it matches or if no specific workflows are defined.
2. `checkSkipPrepID`: Checks if the current workflow's PrepID is in the list of prepIDs to skip, returning False if it should be skipped and True otherwise.
3. `checkRejectedStatus`: Checks if the workflow's request status is one of the rejected statuses, and if it is a RelVal workflow, sets the status to 'forget' and returns False; otherwise, it returns True.
4. `checkPriority`: Ensures the workflow's request priority is above a specified threshold, logging and returning False if it is below the threshold; otherwise, it returns True.
5. `checkWorkflowCount`: Compares the count of pending workflows to the acquired threshold for the workflow's priority block, logging and returning False if the count exceeds the threshold; otherwise, it returns True.
6. `checkRequestStatus`: Checks if the workflow's request status is 'assignment-approved' and sets the workflow to 'away' if it is not, logging the action and returning False; otherwise, it returns True.
7. `checkNextVersion`: Determines the next version of the workflow, setting the status to 'trouble' and returning False if it cannot decide; otherwise, it returns True.
8. `checkOutputTiers`: Ensures the workflow has specified output datasets, logging and returning False if no outputs are found; otherwise, it returns True.
9. `checkGo`: Checks if the workflow controller's 'go' condition is met or if the 'go' option is set, allowing the assignment if either is true.
10. `checkCpuh`: Verifies the workflow's CPU hour requirement against the maximum allowed, logging and returning False if it exceeds the limit and the 'go' option is not set; otherwise, it returns True.
11. `getSites`: Retrieves and sets the allowed and not allowed sites for the workflow, as well as the LHE input, primary, and secondary data required by the workflow.
12. `_getAAAEligibility`: Determines the eligibility for primary and secondary AAA (Any Available Analysis) based on assignment parameters and workflow options.
13. `updateAssignParameters`: Updates the assignment parameters for the workflow by integrating settings from relevant campaigns, ensuring secondary data is allowed, and determining AAA eligibility for primary and secondary data.

14. `updateAllowedSitesAAA`: Modifies the list of allowed sites for primary AAA by removing sites from the allowed list and adding them to the not allowed list based on configuration.
15. `checkSitesAllowed`: Verifies that there are allowed sites for the workflow, logging and returning False if none are found and no site whitelist option is provided.
16. `setLFNvalue`: Sets the Logical File Name (LFN) base for the workflow's primary input data, retrieving it from the database reader for each primary input.
17. `getPrioritySites`: Determines the priority sites for the workflow by selecting Tier-1 and Tier-2 sites from the allowed sites list and choosing a storage element (SE) for output placement.
18. `setParameters`: Sets various parameters for the workflow, including site whitelist and blacklist, non-custodial sites, auto-approve subscription sites, acquisition era, processing string, merged LFN base, processing version, and options for primary and secondary AAA. It also sets the team and adjusts the number of events per job if the workflow requires LHE input.
19. `_uploadRelValPileup`: Placeholder for future implementation of handling RelVal pileup uploads, returning True as a default.
20. `uploadRelValPileup`: Uploads pileup documents to MSPileup for RelVal workflows, logging success or failure and stalling assignment if the upload fails.
21. `_pickOptions`: Updates the parameters dictionary with values from the options dictionary, handling string values with commas by converting them to lists.
22. `_pickCampaign`: Updates the parameters dictionary with values from the assignParameters dictionary, focusing on campaign-specific parameters.
23. `updateParameters`: Updates the workflow parameters by merging options and assignment parameters, prioritizing forced options if specified. Sets the 'execute' flag to True unless in test mode.
24. `checkSplit`: Checks and potentially applies changes to the splitting of the workflow. If changes are on hold and the 'go' option is not set, it logs the status and returns False. If splitting changes are needed, it logs the details and returns True.
25. `processSubscriptionSites`: Updates the workflow parameters to include both non-custodial and auto-approved subscription sites by merging the respective lists and removing duplicates.
26. `getWorkflowParams`: Returns the current workflow parameters.
27. `_heavyWorkflow`: Constructs a notification message for heavy workflows, including the workflow name, output datasets, and assigned sites. It contains a placeholder for sending an email notification.
28. `checkResult`: Evaluates the assignment result. If not in test mode, it updates the workflow status to 'away' upon successful assignment, logs details, and handles heavy workflows. On failure, it logs an error message and sends a critical log to OpenSearch.

10.2 Assignor

A brief description of some functions implemented in Assignor:

1. `parseOptions`: Parses command-line options using optparse, returning a dictionary of options and an optional argument, which represents the first non-option argument if present.
2. `getPendingWorkflowCount`: Retrieves the count of workflows in specific statuses ('assigned', 'staging', 'staged', 'acquired') from the request manager and returns the total count.
3. `_getUnifiedWorkflows`: Fetches workflows from the unified database based on specified statuses or defaults.
4. `_getForceAssignWorkflow`: Creates a workflow instance for force assignment using the specific workflow name. This is to be done when the workflow is not present in the Unified database.
5. `_checkLimit`: Verifies if the total number of assigned and stalled workflows is within the specified limit.

6. `_checkWorkflowCount`: Increments the workflow count, updates the pending workflow count periodically, and checks the workflow count against a threshold using the assignment processor.