

DuraCloud Mill Overview

- [Purpose](#)
- [Queues](#)
- [Queue Monitors](#)
- [Duplication Policies](#)
- [Duplication Policy Editor](#)
- [Looping Duplication Task Producer](#)
- [AuditStorageProvider \(Decorator\)](#)
- [Duplication Task Processor](#)
- [Looping Bit Task Producer](#)
- [Bit Integrity Check Task Processor](#)
- [Worker Manager](#)

Purpose

DuraCloud Mill is massively scalable bulk task processing framework. It performs various tasks including duplication, bit integrity checks, bit integrity report generation, and audit log writing.

Queues

There are 7 queues in the DuraCloud Mill:

1. High Priority Duplication Queue: receives tasks generated by content/space changes in a DuraCloud account which are detected upon processing of AuditTasks.
2. Low Priority Duplication Queue: receives tasks generated by the Looping Duplication Task Producer.
3. Bit Integrity Queue: receives bit integrity check tasks generated by the Looping Bit Task Producer
4. Bit Error Queue: receives bit integrity check errors generated by BitIntegrityCheckTaskProcessors.
5. Bit Report Queue: receives bit report tasks generated by the Looping Bit Task Producer
6. Audit Queue: receives tasks generated by the AuditStorageProvider decorator.
7. Dead Letter Queue: the landing zone for any tasks which cannot be processed, despite multiple attempts. These tasks are reviewed by DuraCloud staff directly.


Queue Monitors

Each queue in the system is monitored for size. All of these monitors are configured in AWS CloudWatch. Both the High Priority Queue and the Low Priority Queue have monitors for high and low queue volume, which are used in combination with Auto Scaling to ensure that the correct number of worker instances are in place to handle the current load. The Dead Letter Queue monitor simply sends notifications when the queue is not empty.

Duplication Policies

Duplication policies define the rules for duplicating spaces across providers. For example, as a DuraCloud account holder I may wish to see all content in spaces A,B, and C in my primary storage provider replicated to a secondary provider. All duplication policies for a single account (ie subdomain) are organized in single json file stored on S3 with the following naming convention:

Duplication Policy Naming Convention

 {subdomain}-duplication-policy.json

Here is an example policy:

Sample duplication policy

```
{
  "spaceDuplicationStorePolicies":{
    "spaceA":
      [
        {"srcStoreId":"1","destStoreId":"2"},
      ],
    "spaceB":
      [
        {"srcStoreId":"1","destStoreId":"2"},
        {"srcStoreId":"1","destStoreId":"3"}
      ]
  }
}
```

In the above example, spaceA would be replicated from storage provider #1 to #2, while spaceB would be replicated to **both** storage provider #2 **and** #3.

The duplication policy files are read by the task producers - ie the Looping Task Producer (LTP) and the DuraStoreTaskProducer (DSTP). It is important to note that both of these task producing processes depend on duplication-accounts.json. This file simply lists the accounts (identified by subdomain) to be considered for duplication and it is found in the same space as the duplication policies themselves. In other words, in order for policies to be read by the task producers there must be a duplication policy file and the subdomain must be referenced in the duplication-accounts.json file. Finally these files can be modified directly themselves. But it is recommended instead to use the Duplication Policy Editor instead.

Example duplication-accounts.json file:

```
[ "account1" , "account2" ]
```

Given this example duplication-accounts.json, two other files would be expected in the same space, named account1-duplication-policy.json and account2-duplication-policy.json.

Duplication Policy Editor

The Duplication Policy Editor (DPE) is responsible, as one might guess from it's name, for creating, updating and deleting duplication policies. The code for the editor is maintained in a [subproject of "duplication" called "policy-editor."](#) It is an HTML/CSS/Javascript app that makes heavy use of the [EmberJS javascript framework](#).

Looping Duplication Task Producer

The [Looping Duplication Task Producer \(LDTP\)](#) is responsible for periodically looping through all duplication policies and placing every eligible space and content item on the low-priority duplication queue. It does not consider whether an item **has been** duplicated; rather it assumes nothing and, consequently, puts everything on the queue. "What does it put on the queue?" you may well ask. The answer is Duplication Tasks. The duplication tasks come in four flavors:

- create a space in the destination storage provider
- delete a space in the destination storage provider
- copy content from the source space to the destination space
- delete content in the destination space.

Hmmm. Not exactly Baskin Robbins, but nevertheless tasty if one happens to be a DuplicationTaskProcessor. For more information on DuplicationTaskProcessors, see the section on the WorkerManager below.

Here's how it works. The LTP is fired off by a cron job every 10 or 20 minutes or so. When it starts, the LTP attempts to read the state file which indicates where it left off after its previous instantiation. If there is no state file, it assumes that the currently executing run is the first time it has been executed. If it is not the first time the LTP was initialized on this machine, the state file will tell the LTP whether or not it stopped in "mid-loop" after the previous instantiation. If it did not stop in mid-loop, then it will check whether or not it is due for another loop. (A loop is a complete run through all the policies associated with all accounts). If it is not due for a loop, it will exit right there. If it is due for a loop it will read all the policies and generate a flat list of "morsels."

Morsels represent the state of a task loop for a single policy. A morsel tracks what has been added to the queue and whether or not deletes have been computed for a single store policy. A store policy is just the smallest operable unit of a duplication policy. As we mentioned above, duplication policy files contain multiple concrete policies which, at their most basic level, define duplication between identically named spaces on two separate storage providers. A morsel tracks what has been added and/or deleted for a given space on a given source storage provider.

So the LTP begins nibbling away at the source spaces, placing a task on the queue for each item in the source space. Before adding duplication tasks for each morsel, first it runs though the destination space and places a delete task on the queue for any item in the destination that does not exist in the source. It is worth noting here that morsels are not generally nibbled to completion all at once. While it will add all deletes onto the queue regardless of the number of items that need to be deleted, in the case of copy tasks, it will only add 1000 items at a time before checking that the maximum task queue size has not been reached. Blocks of 1000 copy tasks are interleaved in this way in order to ensure that all accounts get some duplication love. If no interleaving occurred, it would be possible for a single, very large space to monopolize the low priority queue for hours or days at a time. So anyway, back to the maximum queue size: if queue has reached or passed that limit, the LTP will exit. Once all the morsels have been fully "nibbled," the looping task producer, having both logged the completeness of the run and scheduled the next run, exits.

It may also be worth noting that while all deletions for a single morsel (morsel = space + source + destination), it is not necessarily true that all deletions for a space (across all store policies) will be processed together. The reason for this is simply that after a morsel's deletes are processed, it will first attempt to nibble the first 1000 items before moving onto the next morsel. For example, if there are two store policies associated with one space, the deletions for the first policy will be checked first. If the queue size is still below max, 1000 items would be pulled for duplication. Then deletes would be calculated for the second policy.

Or more precisely the order of events in the above case:

1. if queue < max, spaceA policyA deletes are calculated
2. if queue < max, 1000 dups are added from spaceA, policyA
3. if queue < max, space A policyB deletes are calcuted.
4. if queue < max , 1000 dups are added from space A policy B.
5. (assuming no other spaces) goto 2 then 4
6. repeat 5

AuditStorageProvider (Decorator)

[AuditStorageProvider](#) is a [StorageProvider](#) decorator that intercepts [StorageProvider](#) actions such as content adds, updates, and deletes and,where appropriate, responding to those events by creating audit tasks and dropping them on the audit queue. In turn the [AuditTaskProcessor](#) consults the duplication policies and generates [DuplicationTasks](#) where appropriate and drops them on the High Priority Duplication Queue. Thus, we can expect that a user would wish to see any downstream effects of those actions to be apparent as quickly as possible. Since the low priority queue can potentially take days to process, we have provided a high priority queue which workers will attempt to empty before consuming the low priority queue. The [AuditTaskProcessor](#) does not honor notion of a maximum High Priority Duplication Queue size. The assumption here is that we will always be able to bring sufficient to workers to bear on this queue should the need arise. Additionally, should any items for any reason not get consumed by the Duplication Task Processors (in the event that the workers weren't able to chew through the queue before messages expired) those missed events would be picked up on the next LDTP run.

Duplication Task Processor







The Duplication Task Processor is where all the magic (i.e. duplication) actually happens. When a task processor is created, it is given a single task which indicates the file that it needs to be considered and it is given a source and destination provider. The task processor will first check both the source and destination provider to see if the content item in question exists in both. If the file exists only in the source, a duplication is needed. If the file exists only in the destination, a deletion is needed. If the file exists in both places, the properties of the file are compared. If they do not match exactly, a property duplication is needed. If everything matches, then everything is in order, and no changes need to be made. If an action needs to take place, the call can be made do the destination provider at this point. No changes are ever made on the source provider. All calls to providers occur within a retry loop, to provide a greater assurance of success. Once all of these steps are completed, the work of the task processor is complete.

Looping Bit Task Producer

The Looping Bit Task Processor loops through all content items across all accounts and adds [BitIntegrityCheckTasks](#) to the Bit Integrity Queue, but only after filtering out any items from excluded accounts, stores, or spaces. The exclusions can be defined in a text file and passed into the task producer processor on the command line. The Bit Task Producer will fill the bit integrity queue one space at a time. If there the queue has reached its limit, it will stop and try again when it is invoked again (driven by a chron job). Once it has completed a space, it will then check for an empty queue. Once the queue has emptied, it will create bit integrity report task and then continue to the next space.

Bit Integrity Check Task Processor

Bit integrity check tasks operate on a single content item at a time. It will download the content item, calculate the checksum on the downloaded file, and then compare that value to the storage provider's checksum as well as those stored for the item in the audit log and content-index. The results, pass or fail are recorded in the BitLog database. See the table below for various error conditions, how they might have come about, and how they are resolved.

#	Content	Storage	Content Index	Audit Log	Outcome	How did it happen?
1	 N/A	 	 	 	add bit log item: success	all went as planned

2	✗	✓	✓	✓	add bit log item: failure add item to ResolutionTask queue (may be internally resolvable if secondary store available)	The content went sour
3	✓	✗	✓	✓	add bit log item: failure add item to ResolutionTask queue (externally resolvable: contact storage provider)	The storage provider's checksum process failed
4	N/A	✗	✓	✓	If last retry, wait 5 minutes before trying again If last retry, then generate bit error.	The storage provider's checksum process failed, the audit log is backed up, or an audit task was dropped.
5	✓	✓	✗ or null	✓	add bit log item: failure update content index in place if the audit log properties are null, use storage provider properties to patch audit log item and content index.	The content index was corrupted because an update failed or the checksum itself was corrupted in the process of update
6	✓ or N/A	✓	✓	✗	add bit log item: failure add item to ResolutionTask queue (internally resolvable: audit log out of sync)	The audit log item was corrupted because an insert failed or the checksum itself was corrupted in the process of insertion into Dynamo
7	✓ or N/A	✓	✓	null	add bit log item: failure Add item to audit queue	The audit index was corrupted because an insert failed silently under the AWS covers or the item was manually deleted.
8	404	404	✓	✓	If penultimate retry, wait 5 minutes before putting back on queue. If last retry, then generate bit error.	The item was removed in the Storage Provider, but not captured by DuraCloud (yet)
9	✓	✓	null	null	If penultimate retry, wait 5 minutes before putting back on queue. Otherwise log error and add to the audit queue.	The item was added in the Storage Provider, but not captured by DuraCloud (yet)
10	✓	✓	✗	✗	If penultimate retry, wait 5 minutes before putting back on queue. Otherwise log error and add to the audit queue.	The item was updated in the Storage Provider, but not captured by DuraCloud (yet)
11	404	404	null	null	Do nothing.	Bit integrity processing is behind fully processed deletes.

Worker Manager

The Worker Manager, a.k.a. Workman, is the heart of the system. Or perhaps more aptly, the digestive system. Workman is responsible for managing a pool of worker threads that are in turn responsible for processing different kinds of tasks. Multiple instances of workman may run at the same time, be they on the same and/or separate machines. In fact, the scalability of the system depends on the ability to scale up worker nodes to process queue items in parallel. The workman process attempts to read the high priority queue first, reading up to 10 tasks at a time and distributes them to a pool of workers. If not high priority tasks are available, it attempts to read the low priority queue. Should that queue be empty as well, the DSTP back off exponentially before retrying again, waiting initially for 1 minute and will never wait longer than 8 minutes. Once a task worker thread receives a task to process, it will monitor the progress and make sure that the visibility timeout is extended as necessary to prevent the item from reappearing on the queue. Once the task has been processed, the task worker then deletes the item from the queue. In the case that the task could not be successfully processed, it will be placed at the back of the queue for reprocessing. If it has failed three times, the task will be placed on the Dead Letter Queue for human review.

Now, a key point to consider: Workman can process any number of different kinds of tasks. When a task is passed to a task worker, the worker delegates the creation of a task processor to a task processor factory. The task processor factory is responsible for recognizing tasks and providing an appropriate processor for that task. So as the system develops, new types of tasks can be handled simply by adding a new task processor and registering the related factory with the RootTaskProcessorFactory (see diagram below).