# Dynamic Reservation

A Design Doc for Apache Aurora Epic AURORA-1735

# Motivation

The ability to reserve cluster resources is beneficial to both stateful and stateless services. It provides stateful services with the means to reserve disk space on hosts in order to maintain state and makes an effort to re-schedule them to the same hosts upon failure or after an update. Stateful and stateless services alike receive faster re-scheduling turnover upon a failure by having their spot already reserved and provisioned.

## Persistent Services

As a precursor to support persistent services through Apache Aurora (see AURORA-1714) we have to enable dynamic reservation of resources through Aurora.

A persistent service needs the ability to create a persistent disk volume, and the ability to reserve resources. A persistent volume is necessary to actually store the state, and reservation of resources is necessary in order to reliably re-offer the service: the state itself, along with the resources necessary to retrieve the state (e.g. cpu and memory).

## Mission Critical Services

By granting mission/business critical jobs the privilege of reserving resources and holding on to them, even after they stop running (i.e., for updates or due to failure), Aurora can guarantee faster scheduling turn-over. In large clusters, users might experience slowness in scheduling a large business critical job with multiple instances simply because Aurora is having a hard time finding matching empty slots (potentially after preemption) in the cluster after each update to the job.

Services deployed under containers (e.g., Docker) may have to pull and provision large images for the service to get to the up and running state. By making an effort to re-assign such services to the hosts they were running on before failure, the cluster can prevent already spent resources (e.g., cpu cycles and network bandwidth) from going to waste. This is particularly useful if

software containers are getting deployed as previous image versions have been cached on the reserved host; very useful when aborting a bad deploy already in progress.

A bad code deployment might trigger a rollback during a job update. Currently, we are paying double the scheduling/provisioning cost; once for the task to be killed and rescheduled for upgrade and another time to be killed and rescheduled for downgrade. Reserved resources can significantly improve the rescheduling overhead during job update failures.

# User Stories

## Create and run a job

As a cluster user, I want to create and run a job with already reserved resources.

### Implementation

For services with require dynamic reservations `OfferManager.launchTask` will call `SchedulerDriverService`which will expose `acceptOffers` inside (this was landed) which we will reserve the offer and launch the tasks. Existing call `drive.launchTask` will stay the same for non-dynamically reserved jobs. Reservation will use labels other than ~~TaskId~~.

## Kill a job

As a cluster user, I want to kill all instances of my running job and unreserve its resources.

### Implementation

`SchedulerDriverService` will keep on using `killTask` function which will call  and will be used by `StateManagerImpl.updateTaskAndExternalState` in `case KILL`.

Proposed change is to add another abstraction (OfferReconciler) which will have OfferManager and Storage as it's dependencies. Every incoming offer goes through `addOffer` in `OfferManager` which will add an OfferAdded event that will get posted to the EventBus.

OfferReconciler then will subscribe to those events, and then for each offer look through labels which have jobkey/instance_id. Then we try to pull back any active tasks matching that jobkey/instance_id. If we find one, then we do nothing (why did we get that offer in the first place?). If we find no matches then we will unreserve the offer.

OfferManager will add a new function `unreserveOffer` which will accept an offerId and List<Protos.Resource>. It will loop over the list and set correct attributes to unreserve them, and will use `acceptOffers` interface to send the Unreserve operation to Mesos master.

There could be performance concerns we need to address as for every offer we need to look through all of them and match the labeled taskId against the TaskStore.

## Add instances to a job

As a cluster user, I want to add new instances to my running job that is using reserved resources.

### Implementation

This should have no effect and will follow the Create and run a job and Kill a job flows. All existing tasks should be killed and will be unreserved on the next run.

## Update a job

As a cluster user, I want to update the configuration of an already running job that uses reserved resources.

- I may choose to roll out updates to my job following the canary deployment pattern.
- My job update attempt might fail and result in an automated rollback.

### Implementation

If an existing offer which was previously dynamically reserved can not satisfy the ResourceRequest, then this offer will be vetoed and another offer will be reserved. It will be the job of OfferReconciler to unreserve the reservation which is no longer needed.

## Restart a job

As a cluster user, I want to restart instances of a job using reserved instances without unreserving their resources.

### Implementation

No changes.

## Enforce quota

As a cluster operator, I want to enforce role quota by including reserved resources.

Implementation

No changes.

# Design Decisions

## Dynamic Reservation Scope

The scope of non-reserved resources is limited to the lifetime of the holding task. Once a task finishes or dies the resources are automatically freed. Dynamically reserved resources will remain reserved during task fail-overs. We envision the following two scopes for dynamically reserved resources. However, this MVP seeks to realize dynamically reserved resources per task (option A).

### Option A: Dynamically Reserved Resources Per Task

- Resources are automatically reserved/unreserved for a single task.
- Reservation is made right before the task is launched.
- When a task is finished or killed the resources are automatically unreserved.
- Provides faster re-scheduling upon task failure (the failure should not make the reservation offer go away). Has no impact on scheduling.
- Reservations need not to be persisted (can be inferred from a combination of tasks and offers).
- Job updates result in automatic reservation changes.

### Option B: Dynamically Reserved Resources Per Role

- Resources are manually reserved/unreserved for a role.
- Reservation can be made in advance before any job is scheduled (for example, 4 cpus, 8g ram, 128g disk is reserved on 100 cluster hosts).
- Multiple jobs can be scheduled using the same reservation.
- Reservation is kept even after the task is finished or killed.
- The reservation concept replaces Aurora resource quota.
- Provides faster scheduling (as long as the role has enough headroom in quota to accommodate tasks).
- Reservations are persisted (can be computationally expensive to infer from the combination of tasks and offers).
- Job updates can be done without modifying reservations.

## Job Configuration

- Jobs are configured to consume reserved resources through the `tier` attribute of the `Job` object.
- In order to be able to consume reserved resources jobs have to be assigned a tier with `reserved=true`.
- A new boolean tier attribute `reserved` with values `True` and `False` will be added to tier definitions.
- A new tier named `reserved` will be added to `tiers.json` with these attributes: **"reserved"**: {**"preemptible"**: **false**, **"revocable"**: **false**, **"reserved"**: **true**}.
- The combination of tier attributes `preemptible=true` and `reserved=true` is not allowed. A task is preempted for its resources to be assigned to other higher priority tasks. Reserved resources are not taken away from a task after it is preempted hence the combination of the two traits (`preemptible` and `reserved`) is not supported.
- Mesos does not yet support dynamic reservation of revocable resources. Therefore, creating a job with the combination of tier attributes `reserved=true` and `revocable=true` is not allowed.

## Labeling Reservations

Mesos resource reservations do not offer a unique identifier like reservation ID to allow matching reservation requests and offers (it was discussed in Dynamic Reservation Design Doc and there is an accepted Mesos ticket for it). Meanwhile, matching is expected to be performed by comparing the resources themselves. Although still possible, it can potentially be slow for large clusters, and cause implementation complications when Mesos merges multiple resource reservations on the same agent into one when offering them to Aurora. For reference and future, there is a design doc for adding reservation id support to Mesos, but it was deferred in favor of labels documented here.

Mesos allows supplying an optional list of key-value pairs upon reservation of resources that prevents such resource offers from being merged if the assigned key-value pairs do not match (see **Labeled Reservations** section in Apache Mesos Documentation: Reservation). Aurora *job key* (`ROLE/ENV/NAME`) and task *instance ID* are stored as key-value pairs under `reservation` field to help identify the task instance using it.

According to Mesos dynamic reservation documentation, two or more reservation offers on the same host can be merged by Mesos master to form a larger offer:

> "If two dynamic reservations are made for the same role at a single slave (using the same labels, if any; see below), the reservations will be combined by adding together the resources reserved by each request. This will result in a single reserved resource at the slave."

This can result in unexpected behavior when Aurora attempts to reserve two or more resource bags on the same host to launch instances of the same job. The combination of *job key* and *instance ID* under `labels` field forces reservations to be globally unique.

> "Note that two reservations with different labels will not be combined together into a single reservation, even if the reservations are at the same slave and use the same role."

For example, two reservations $R_1$ and $R_2$ for the job `foo/bar/hello` on the same host are labeled uniquely and will not be merged.

$R_1$:

```
[
  {"key": "job", "value": "foo/bar/hello"},
  {"key": "instance", "value": "1"}
]
```

$R_2$:

```
[
  {"key": "job", "value": "foo/bar/hello"},
  {"key": "instance", "value": "2"}
]
```

**TODO:** add random value under labels to prevent merging when reservations of the same instances land on the same host. For example, during a job update with resource reshaping, both reservations of the old and new job might be made on the same host.

## Best Effort

Dynamic reservation is a best effort strategy; in presence of failure, a reservation previously made is valid as long as the offer can be relayed from the Mesos slave to Mesos master and from there to Aurora. If a failure breaks any link in the above communication channel, Aurora would not know of and cannot take advantage of the reservations already made.

Consequently, in the event of host and network failures, scheduler is forced to reserve and reschedule a task on a new host when the original reservation goes away. This results in loss of persistent state, re-provisioning and slower re-scheduling. See Future Work: Persistent Volumes section for a discussion of how dynamic reservations need to be modified in the future to support persistent volumes.

# Unreserving Resources

## Killed Tasks

A task might get lost or killed and respawned; at which point a replacement task gets rescheduled to take its place. If a task has resources reserved, they will not be unreserved because they can be used by the replacement task. However, when a user chooses to kill a task instance through `aurora job killall` command, reserved resources have to be unreserved.

## Updated Tasks

`JobUpdateController` keeps watching all task updates in progress until they transition to success/fail state. When an instance $T_1$ is killed so that a task $T_2$ with the same resource shape to be launched, Aurora should refrain from reclaiming $T_1$'s reserved resources when it kills $T_1$; $T_2$ can reuse $T_1$ reservation. However, if $T_1$ and $T_2$ require a different shape of resources, reserved resources cannot be reused and should be unreserved.

**TODO:** How does `TaskStateMachine` differentiate between the following scenarios of killing a task?

1. Task is killed for update when resources are not reshaped.
2. Task is killed for update when resources are reshaped.
3. Task is killed forever (e.g. through `aurora job killall`).

In case 1, reserved resources should be left behind while in case 2 and 3 they should be unreserved once the task finally transitions to `DELETED` state.

# Handling Operation Failures

According to Mesos documentation:

> "Attempts to dynamically reserve resources or create persistent volumes might fail—for example, because the network message containing the operation did not reach the master or because the master rejected the operation. Applications should be prepared to detect failures and correct for them (e.g., by retrying the operation)."

Aurora task reconciliation process uses time-outs to identify this type of failures; if a task remains in a transient state like `ASSIGNED` beyond the tolerable threshold of `transient_task_state_timeout` (=5 minutes by default) it is marked `LOST` and gets automatically respawned.

When Aurora attempts to dynamically `RESERVE` resources and `LAUNCH` a task (using a single `acceptOffers()` call) the outcome can be one of the following three scenarios:

1. Task sends status update before timeout period, hence Aurora realizes that both operations succeeded (no op).
2. Task does not transition out of `ASSIGNED` state and `TaskTimeout` marks it as `LOST`. When handling the replacement task
    a. `TaskAssigner` finds the matching reserved resource offer from the previous round, hence it makes another attempt to `LAUNCH` the task using the existing reserved resources (i.e., the `RESERVE` operation has succeeded but the `LAUNCH` operation might have failed).
    b. `TaskAssigner` does not find a matching reserved resource offer, hence it makes another attempt to both `RESERVE` resources and `LAUNCH` the task (i.e., both `RESERVE` and `LAUNCH` operations might have failed and need to be retried).

## Reservation Reconciler

The reservation reconciler handles excess reservation offers arising from `UNRESERVE` operation failures. Alternatively, when a host *H* on which resources are already reserved suddenly becomes unavailable (for example, due to network failure or operator initiated maintenance) Aurora needs to dynamically reserve other resources on a new host *H'* to reschedule lost tasks.

Later, when *H* re-registers with the Mesos master and re-offers reserved resources $O_H$, Aurora reservation reconciler has to detect and handle this excess inventory of reservations that is no longer needed.

**Assumption:** Reservation resource offers are labeled by *instance key* (i.e., the combination of *job key* and *instance ID*).
**Definition:** An *unassigned* reserved resource offer is a resource offered to Aurora scheduler by Mesos master that has been previously reserved but currently is not assigned to a task. This can be a resource that has not been assigned to a task since it was reserved, or its previously assigned task is `LOST`, `KILLED`, `FAILED`, or `FINISHED`.

```
let reservations: tasks → unassigned reserved resource offers
for each task T in reservations do
 let Offers_T be the set of unassigned reserved resource offers for T
 if |Offers_T| > 0 then
  if T is dead then
    unreserve all offers in Offers_T
   else if |Offers_T| > 1 then
    unreserve |Offers_T| - 1 offers in Offers_T
```

Algorithm: Reconciling over-reservations

**TODO:** During a job update involving resource reshaping Aurora might see two reservations for the same job instance (old and new configurations). How does reservation reconciler know to

unreserve the right reservation? Perhaps by comparing the shape of reserved resources with job configuration?

# Job Updates

Dynamic reservation speeds up the rate at which instances are restarted during update. However, trying to hold on to reserved resources during various job update scenarios can greatly increase the implementation complexity of the dynamic reservation feature. Considering their impact on dynamically reserved resources, job updates can be divided into the following scenarios:

1. Instance is removed: Aurora automatically unreserves resources after instance is killed.
2. Instance is added: Aurora automatically reserves resources before launching a new instance of the job.
3. Job configuration is changed and
    a. Resources are not reshaped: Resources remain reserved.
    b. Resource tier is modified (from non-reserved to reserved and vice versa): Resources are reserved/unreserved with each shard restart.
    c. Resources are reshaped: Not supported. Otherwise we need to deal with modifying reshaping task resources in place or reserving new resources when scale up is not possible.

In scenario 3c above, reserved resources are unreserved and new resources are reserved with each shard restart. In the future, we might come up with a more efficient way of handling this unsupported scenario. For example, if scaling up and there are additional resources that can be reserved on the same host then additional resources are reserved and merged with the existing resources. Similarly, if scaling down, a partial unreserve frees up the excess reserved resources.

# Tracking Dynamically Reserved Resources

Aurora caches but does not persist Mesos resource offers. As a result, upon scheduler failover Aurora retrieves offers from Mesos master and rebuilds its internal cache. Aurora can follow a similar approach with dynamically reserved resources (Option A) or persist dynamically reserved resources (Option B). This MVP seeks to keep track of dynamically reserved resources via in-memory cache data structures (Option A).

## Option A: Dynamic Reservations Are Cached By Aurora

- If an agent becomes unreachable, Aurora would not know about its reservations. In other words, Aurora would be aware of resources reserved on a host if it (a) sees offers originating from that host or (b) is aware of active tasks consuming reserved resources assigned to that host.

- Mesos resource offers give the dynamic reservations made but yet not assigned to any tasks. Similar to non-reserved resource offers, after a reserved resource offer is accepted by Aurora, to launch a task, it will be taken off the Mesos offers list.
- Aurora maintains an internal cache of cluster state through which it knows of tasks running on Mesos agents. Through this internal cache, Aurora knows of all reserved resources held by tasks on agents across the cluster.
- Aurora reserves additional resources if it finds unassigned tasks for which it cannot see a matching reservation offer. It does not know of reserved resources on unreachable Mesos agents, therefore it cannot wait for them to be offered back in the future.

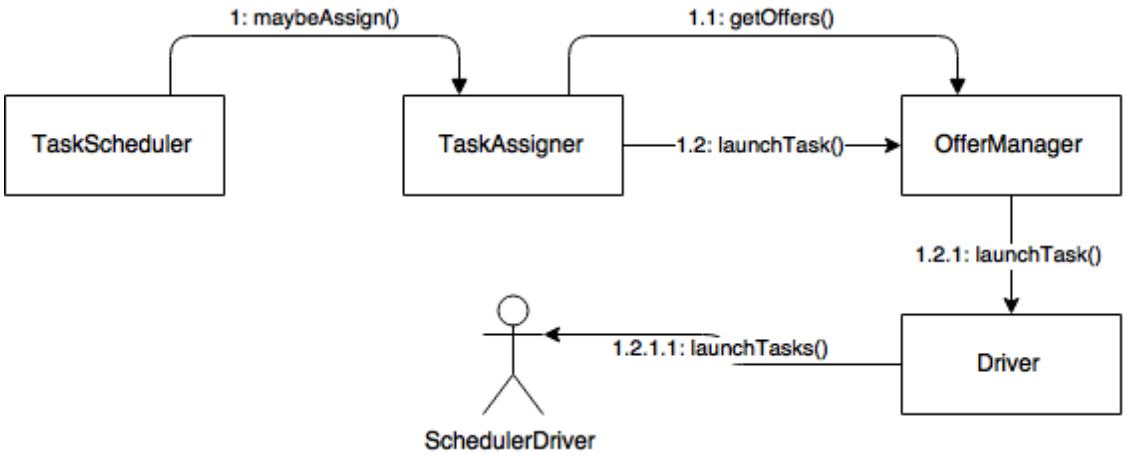
### Option B: Dynamic Reservations Are Persisted By Aurora

- Aurora has a record of dynamic reservations even after hosts become unreachable.
- A time-out mechanism can be used before making new reservations when existing reserved resources become unreachable beyond an acceptable threshold.
- Aurora persistent storage is used to query dynamic reservations.

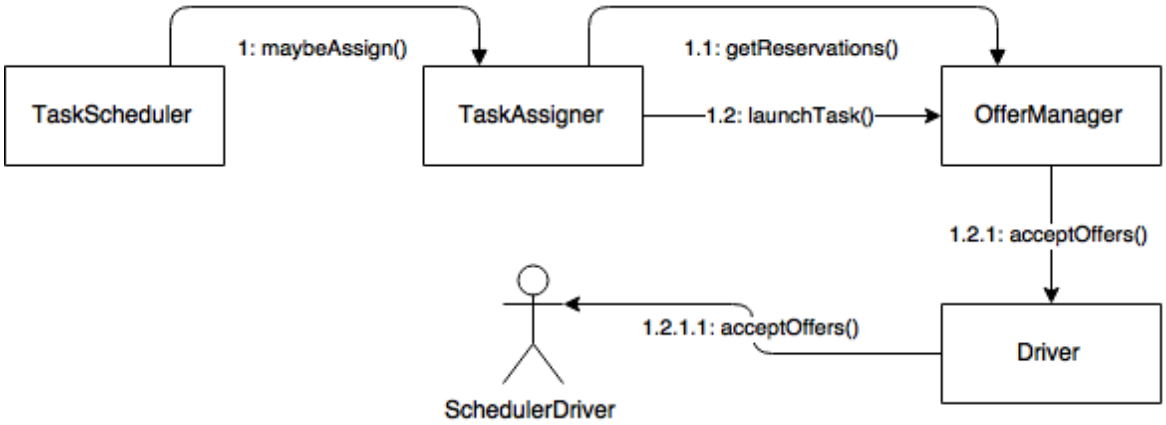
## AuroraExecutor Overhead

Aurora factors in additional resources for AuroraExecutor when accepting offers to launch a task (if it is configured to run with Aurora's custom executor). By default, 0.25 cpu and 128 MB mem is added to the task resources. Similarly, when reserving resources for a task, the executor overhead (if any) needs to be factored in. For example, a task requesting 1 cpu, 128M ram, and 128M disk reserves 1.25 cpu, 256M ram, and 128M disk per instance.

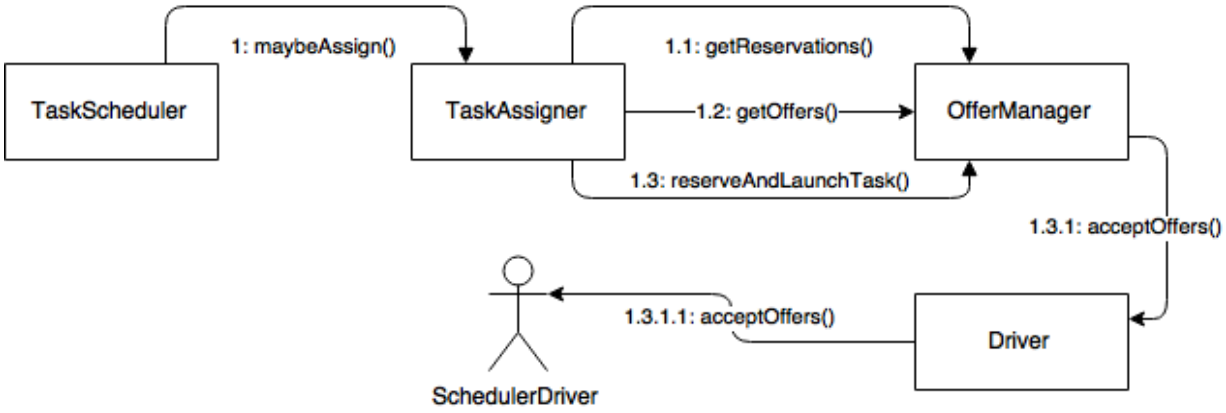
# Proposed Solution



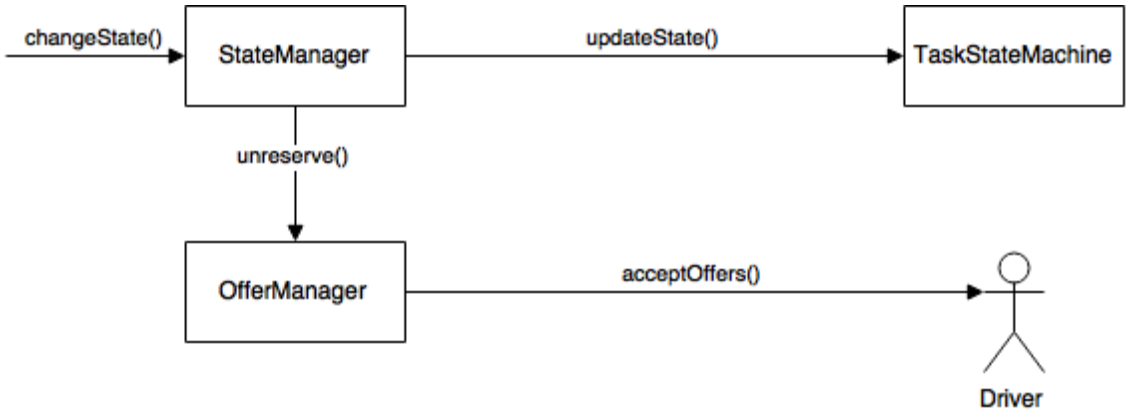

Existing behavior: Launching a `PENDING/THROTTLED` task with a matching resource offer



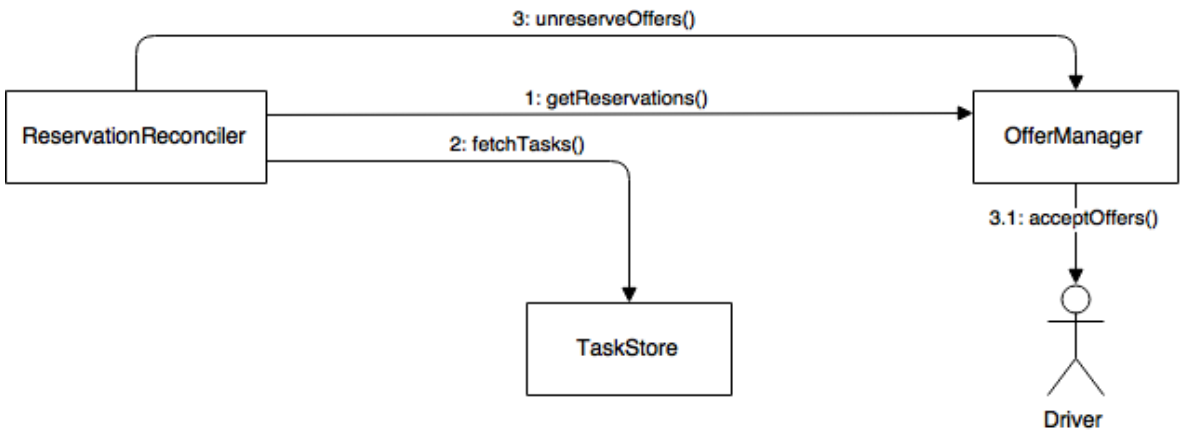

New behavior: Launching a `PENDING/THROTTLED` task with an existing reservation offer




New behavior: Reserving resources and launching a `PENDING/THROTTLED` task when a matching reservation offer is not available




New behavior: Unreserving resources when task is killed




New behavior: Unreserving excess reserved resources

# Decision Required

## Cron Jobs

Anyone see a strong use case for supporting dynamically reserved resources with cron jobs? If not, this can be postponed until the need materializes.

We envision the following two strategies for handling reserved resources consumed by cron jobs.

### Option A: Dynamically Reserved Resources Per Job

- Resource reservations are made after a cron job is scheduled.
- Resources remain reserved until cron job is descheduled.
- Results in wasted cluster resources for sizable cron jobs (with many instances) that are scheduled to run infrequently (e.g., once a day).
- Provides better scheduling performance for cron jobs scheduled to run frequently (e.g., every minute).

### Option B: Dynamically Reserved Resources Per Run

- Resource reservations are made after each trigger of the cron job.
- Resources are unreserved after each run of the cron job.

# Future Work

## Persistent Volumes

Dynamic reservations are designed primarily with stateless jobs in mind. In the future, when Aurora provides support for persistent volumes, we might need to make changes to reserve/unreserve workflows. For example, if a persistent volume is created using a dynamic reservation, users might prefer to have the service instance blocked once the host goes away and reserved resources become unavailable. Relaunching a stateful service instance on a new host without providing it with access to the data in its persistent volume may not be acceptable.

Aurora would need to treat reserved disk resources associated with a persistent volume differently; it would not immediately fall back to reserving new disk resources once an existing reserved disk becomes unavailable. Aurora can optionally refrain from unreserving existing resources and destroying a persistent volume until an operator recovers persisted state from the old disk and restores it in a new persistent volume.

This in turn requires Aurora to persist persistent volumes so that it can differentiate between a volume that has been previously created (i.e., disk resources have already been reserved) and that which is just to be created (i.e., disk resources should be reserved).

# References


Apache Mesos Documentation: Reservation
Apache Mesos Documentation: Persistent Volumes
Dynamic Reservation Design Doc
Reservation Labels Design Doc
Reservation IDs Design Doc
Dmitriy's POC for the Dynamic Reservation