Joe Smith - @Yasumoto

Tech Lead, Aurora and Mesos SRE, Twitter

Hello everyone, welcome to the last slot of the day!

I’m Joe Smith, and I’ve been running the Aurora and Mesos clusters at Twitter for almost 3.5 years now

SLA-Aware Maintenance for Operators

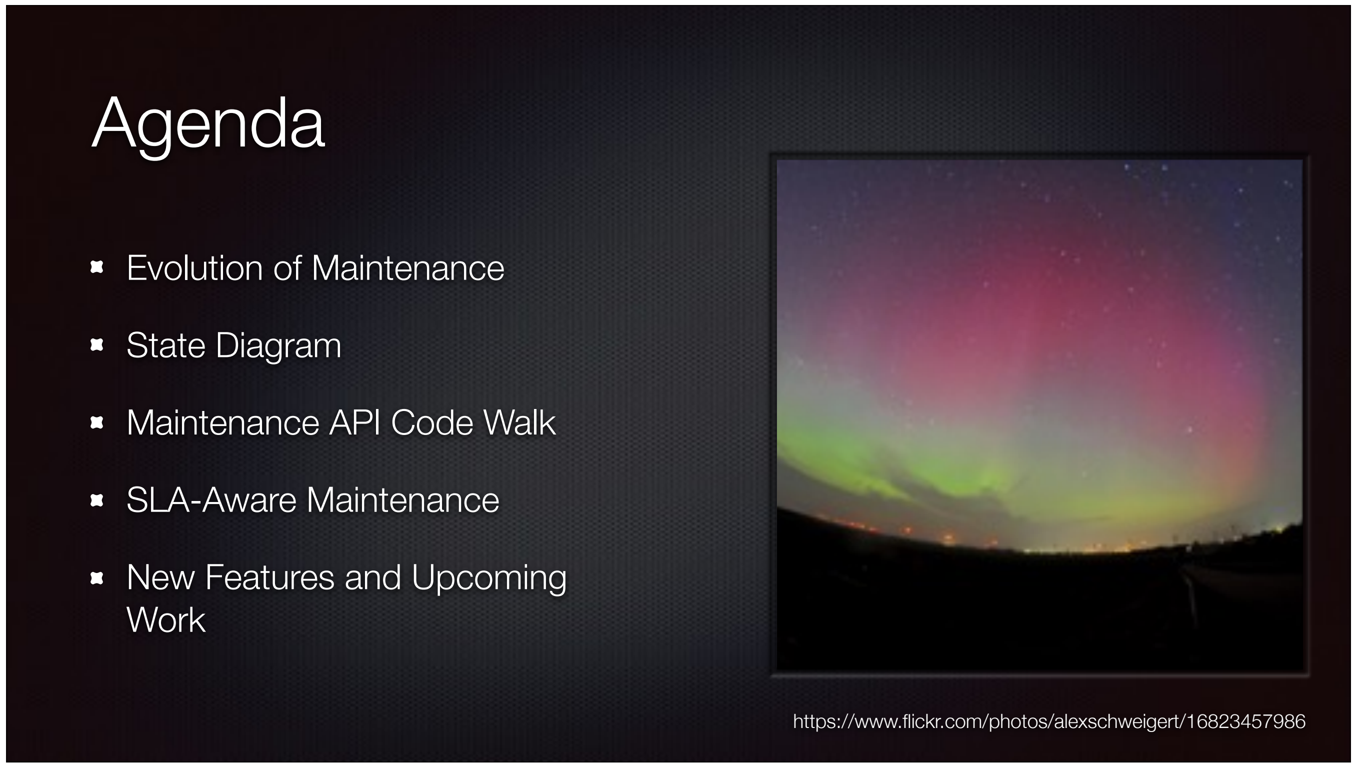
Operations with Apache Aurora and Apache Mesos

This is part of a series of talks I’ve given about Aurora and Mesos from the SRE/DevOps perspective.

The first was a huge information dump of how we build, test, and deploy Aurora and Mesos.

The second, at MesosCon in Seattle this year, described the situations and errors we’ve seen in production, as well as techniques for avoiding them or getting out of trouble.

This one is what I consider the next phase.. you're already running Mesos and Aurora, how do you upgrade?



Agenda

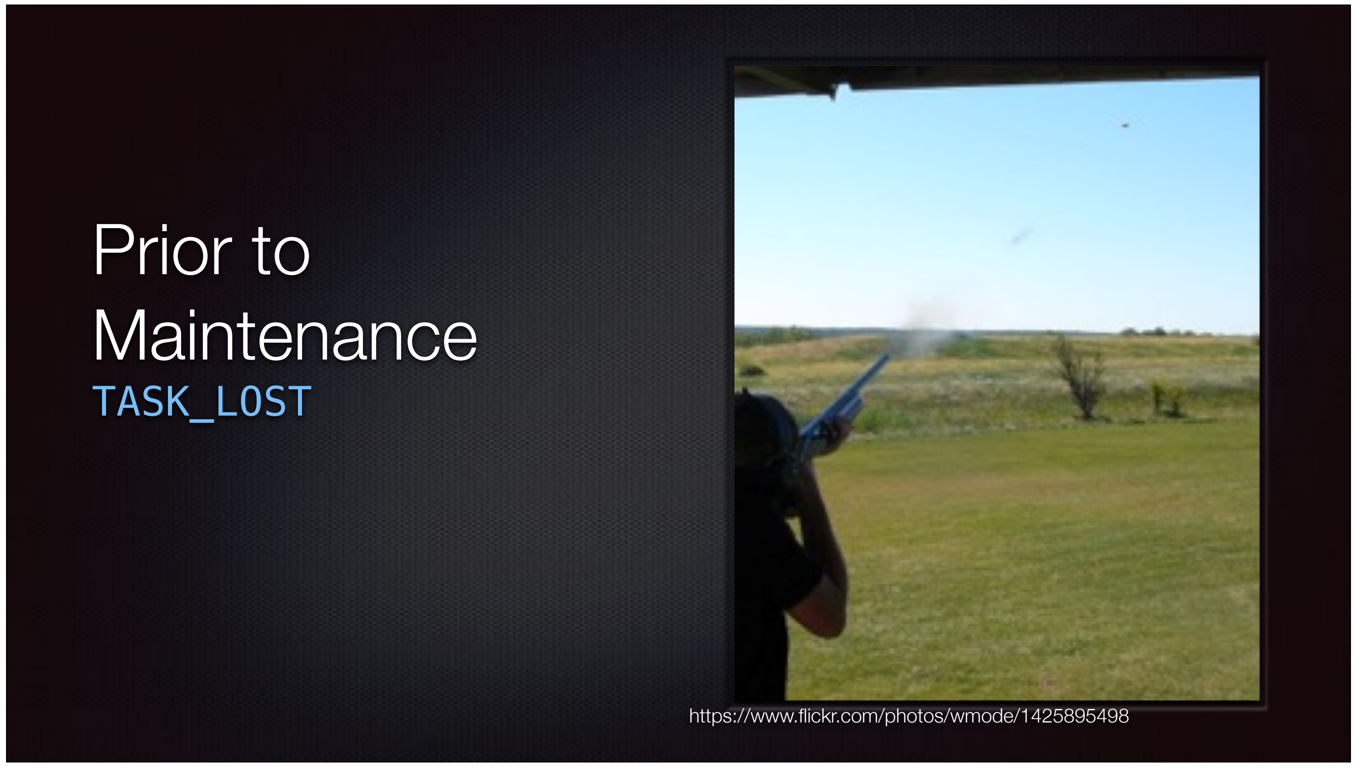
- Evolution of Maintenance
- State Diagram
- Maintenance API Code Walk
- SLA-Aware Maintenance
- New Features and Upcoming Work

https://www.flickr.com/photos/alexschweigert/16823457986

We'll start off with how we got here- before any maintenance primitives at all. This caused lots of cluster churn as tasks disappeared, and our users were very confused. We'll do a bit of code walking to see how that message is transferred through the stack as well.

After that, we'll go over the high-level overview of the general maintenance primitives- then dig into what has actually enabled us to move quickly with our infrastructure, Aurora's SLA for Jobs.

Lastly, we'll touch on two pieces, Mesos' Maintenance- in 0.25.0! and determining how to implement custom SLAs in Aurora, which will help us continue to improve operations.



Prior to Maintenance

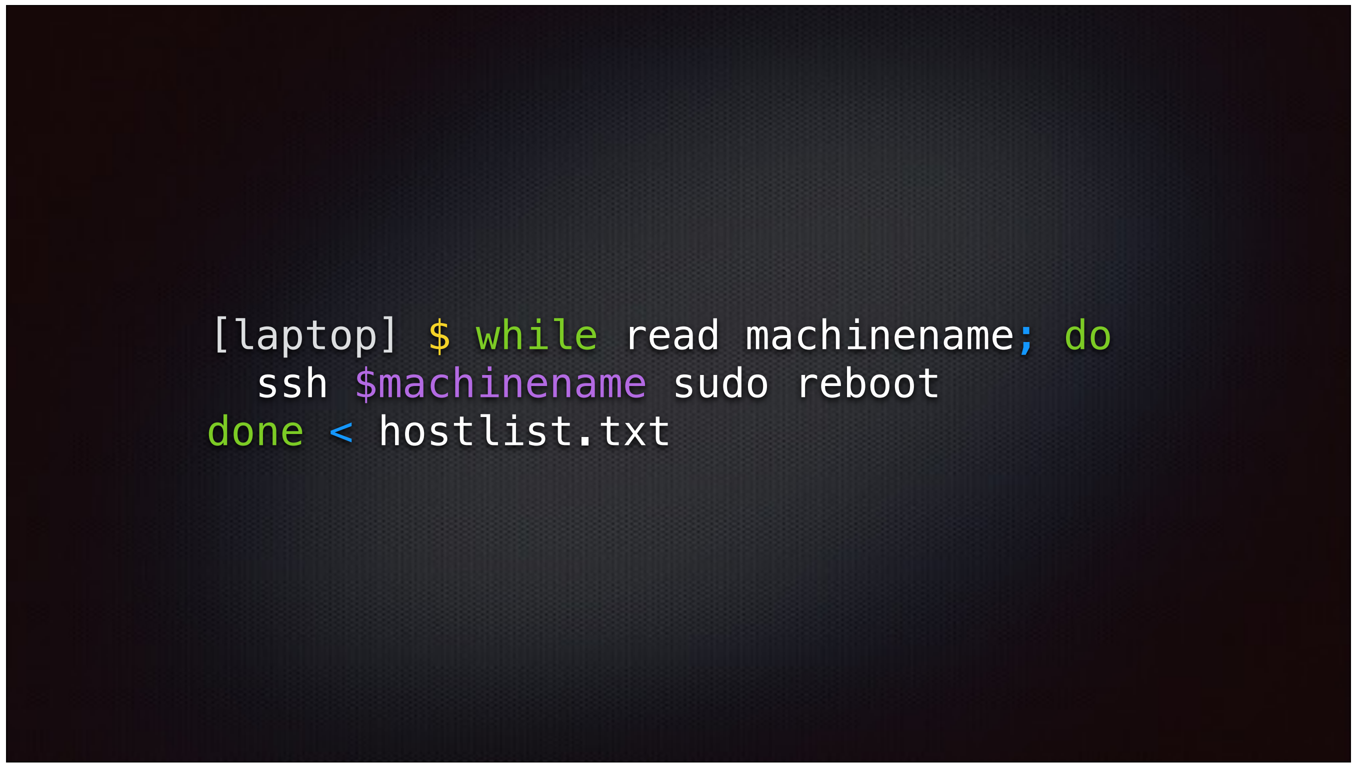
`TASK_LOST`

https://www.flickr.com/photos/wmode/1425895498

So let's start off by walking through how we got here.

When dealing with small sets of host, you can treat each one individually:
  Take aim, breathe, and perform your operation

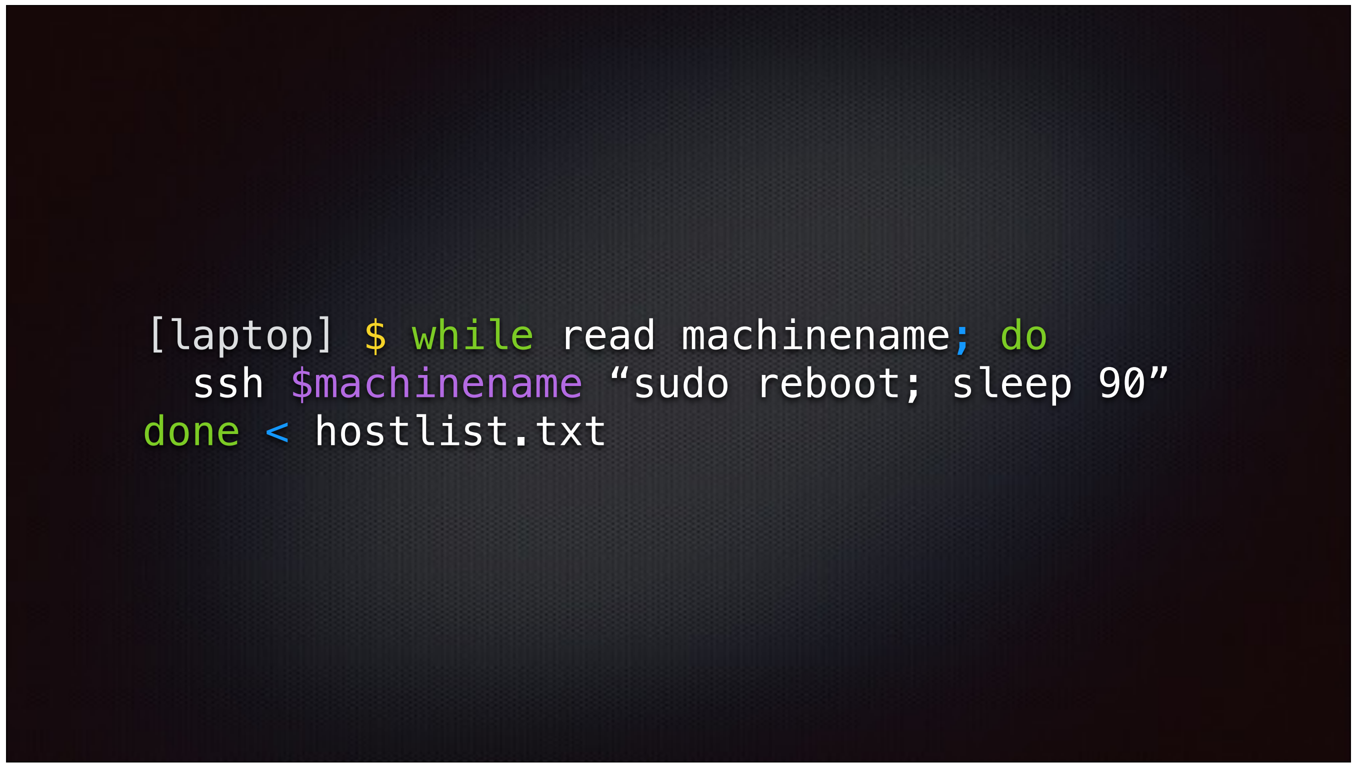
This might be ssh-ing into each server and rebooting it, waiting for it to come back up, then ssh-ing back in.



```
[laptop] $ while read machinename; do
  ssh $machinename sudo reboot
done < hostlist.txt
```

To move from a SysAdmin to #devops… we automate a bit

Again.. this was years ago, and the cluster was relatively small



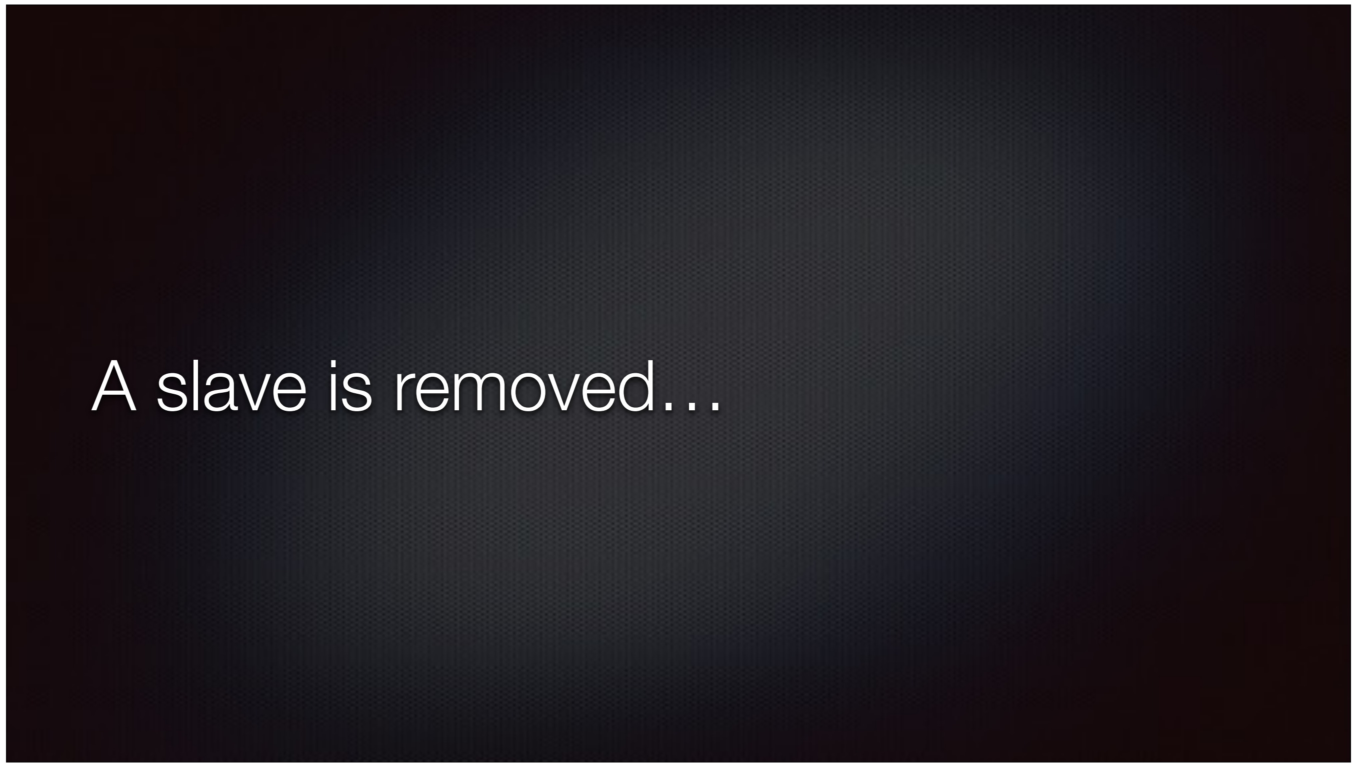
```
[laptop] $ while read machinename; do
  ssh $machinename “sudo reboot; sleep 90”
done < hostlist.txt
```

We were maybe a little bit more advanced….

But really, we have no understanding of how we’re impacting our users when we do this

When you have a larger fleet of machines, especially if they're a relatively homogeneous set, you can treat them the same.

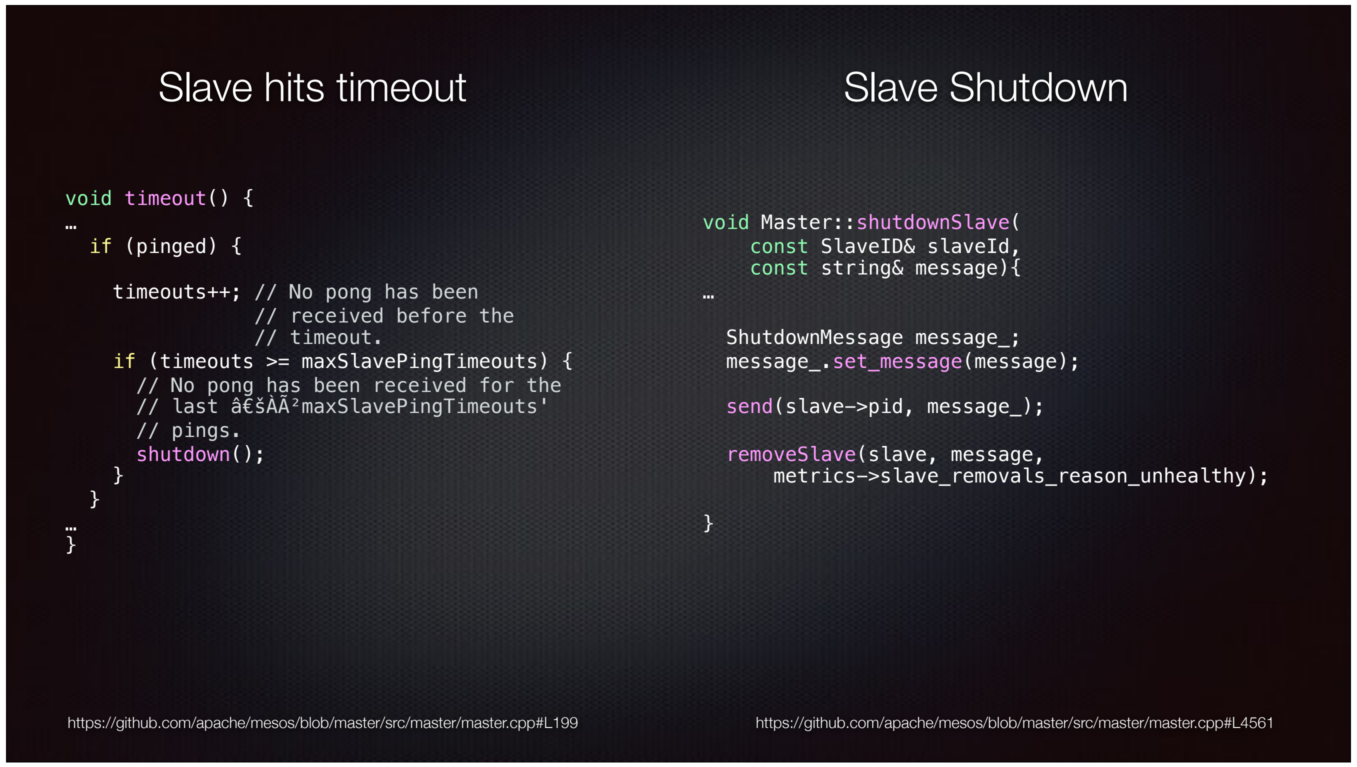
This was the state of maintenance before any tooling- essentially we would just creep across the cluster, rebooting/reimaging/restarting Agents without worrying about the damage we'd do to user tasks



A slave is removed…

So what happens when you lost a slave?

When you’re running these components- core, foundational infrastructure, it’s very helpful to be bold and dig into the code to really understand what’s happening. This means you can be prepared when it breaks.



Slave hits timeout

```
void timeout() {
…
  if (pinged) {

    timeouts++; // No pong has been
                // received before the
                // timeout.
    if (timeouts >= maxSlavePingTimeouts) {
      // No pong has been received for the
      // last â€šÀÃ²maxSlavePingTimeouts'
      // pings.
      shutdown();
    }
  }
…
}
```

https://github.com/apache/mesos/blob/master/src/master/master.cpp#L199

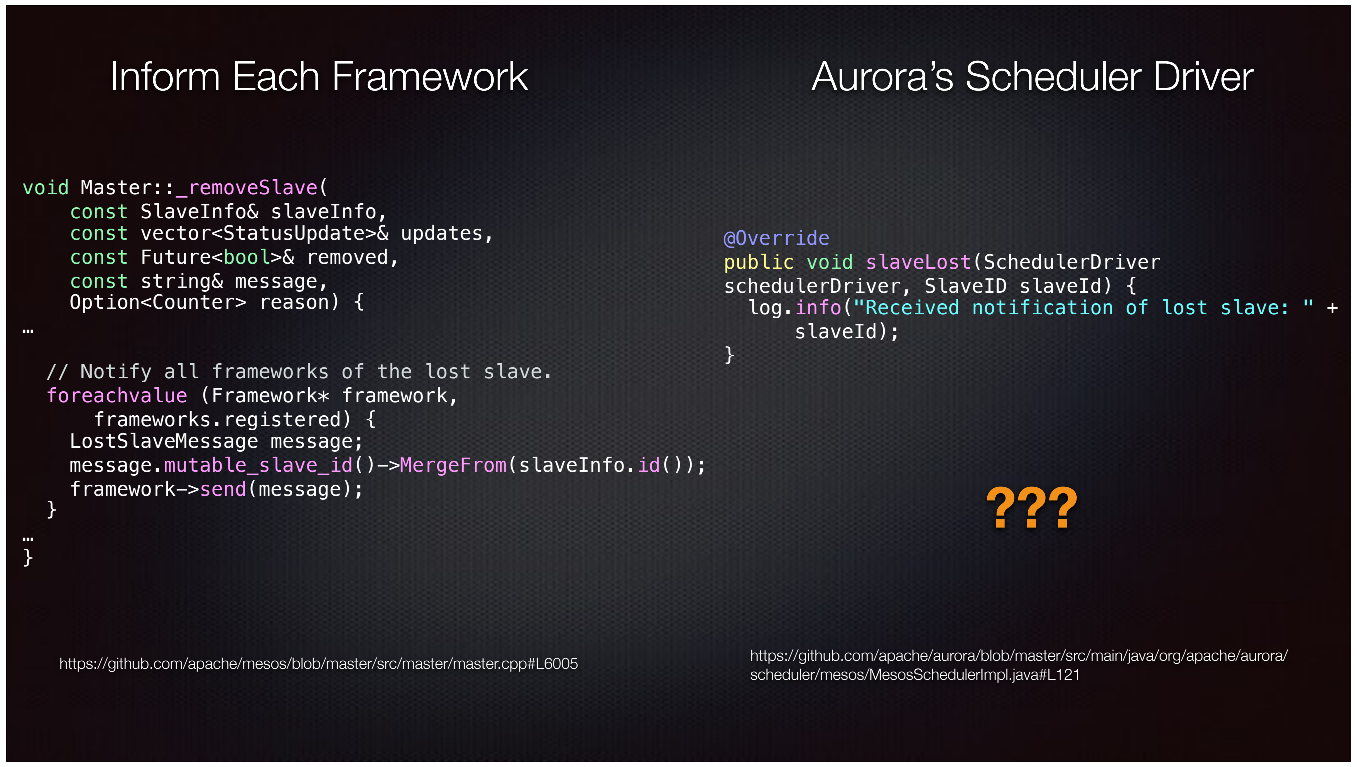
Slave Shutdown

```
void Master::shutdownSlave(
    const SlaveID& slaveId,
    const string& message){
…

  ShutdownMessage message_;
  message_.set_message(message);

  send(slave->pid, message_);

  removeSlave(slave, message,
      metrics->slave_removals_reason_unhealthy);

}
```

https://github.com/apache/mesos/blob/master/src/master/master.cpp#L4561

The master has a health check which agents must respond to.

If the master doesn't hear back after sending a number of pings, it will need to assume that something Bad™ happened to the slave, and it has gone away.



Inform Each Framework

```
void Master::_removeSlave(
    const SlaveInfo& slaveInfo,
    const vector<StatusUpdate>& updates,
    const Future<bool>& removed,
    const string& message,
    Option<Counter> reason) {
…

  // Notify all frameworks of the lost slave.
  foreachvalue (Framework* framework,
      frameworks.registered) {
    LostSlaveMessage message;
    message.mutable_slave_id()->MergeFrom(slaveInfo.id());
    framework->send(message);
  }
…
}
```

https://github.com/apache/mesos/blob/master/src/master/master.cpp#L6005

Aurora's Scheduler Driver

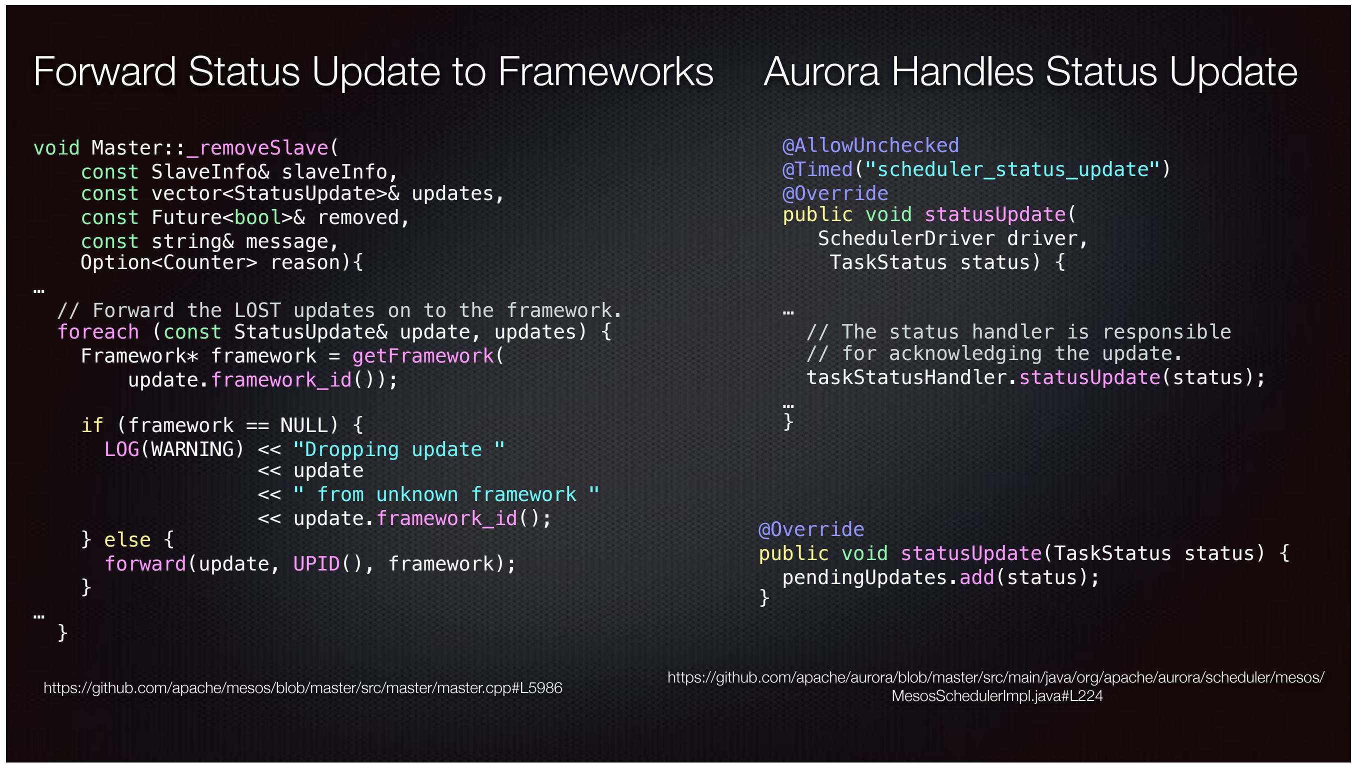
```
@Override
public void slaveLost(SchedulerDriver
schedulerDriver, SlaveID slaveId) {
  log.info("Received notification of lost slave: " +
      slaveId);
}
```

???

https://github.com/apache/aurora/blob/master/src/main/java/org/apache/aurora/scheduler/mesos/MesosSchedulerImpl.java#L121

It then needs to let each registered framework know about the missing agent.

HOWEVER… Aurora doesn't do anything?! Let's move up a few lines in `_removeSlave`.



Forward Status Update to Frameworks

```
void Master::_removeSlave(
    const SlaveInfo& slaveInfo,
    const vector<StatusUpdate>& updates,
    const Future<bool>& removed,
    const string& message,
    Option<Counter> reason){
…
  // Forward the LOST updates on to the framework.
  foreach (const StatusUpdate& update, updates) {
    Framework* framework = getFramework(
        update.framework_id());

    if (framework == NULL) {
      LOG(WARNING) << "Dropping update "
                   << update
                   << " from unknown framework "
                   << update.framework_id();
    } else {
      forward(update, UPID(), framework);
    }
…
  }
```

https://github.com/apache/mesos/blob/master/src/master/master.cpp#L5986

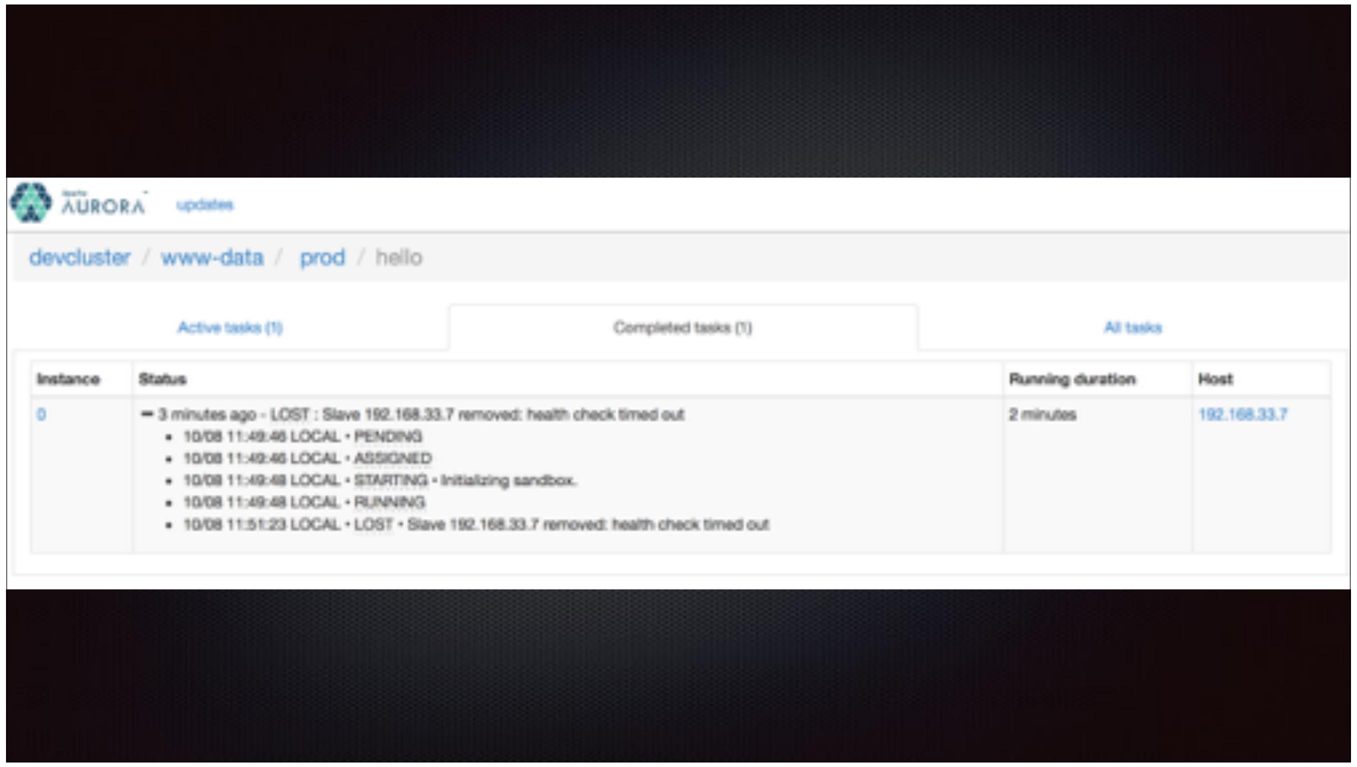
Aurora Handles Status Update

```
  @AllowUnchecked
  @Timed("scheduler_status_update")
  @Override
  public void statusUpdate(
     SchedulerDriver driver,
      TaskStatus status) {

  …
    // The status handler is responsible
    // for acknowledging the update.
    taskStatusHandler.statusUpdate(status);
  …
  }



@Override
public void statusUpdate(TaskStatus status) {
  pendingUpdates.add(status);
}
```

https://github.com/apache/aurora/blob/master/src/main/java/org/apache/aurora/scheduler/mesos/MesosSchedulerImpl.java#L224

Here we see that the Master also informs each framework about the LOST tasks on those machines.

THIS is what Aurora uses to determine if a task has gone away, and it will reschedule that task if it belongs to a Service.



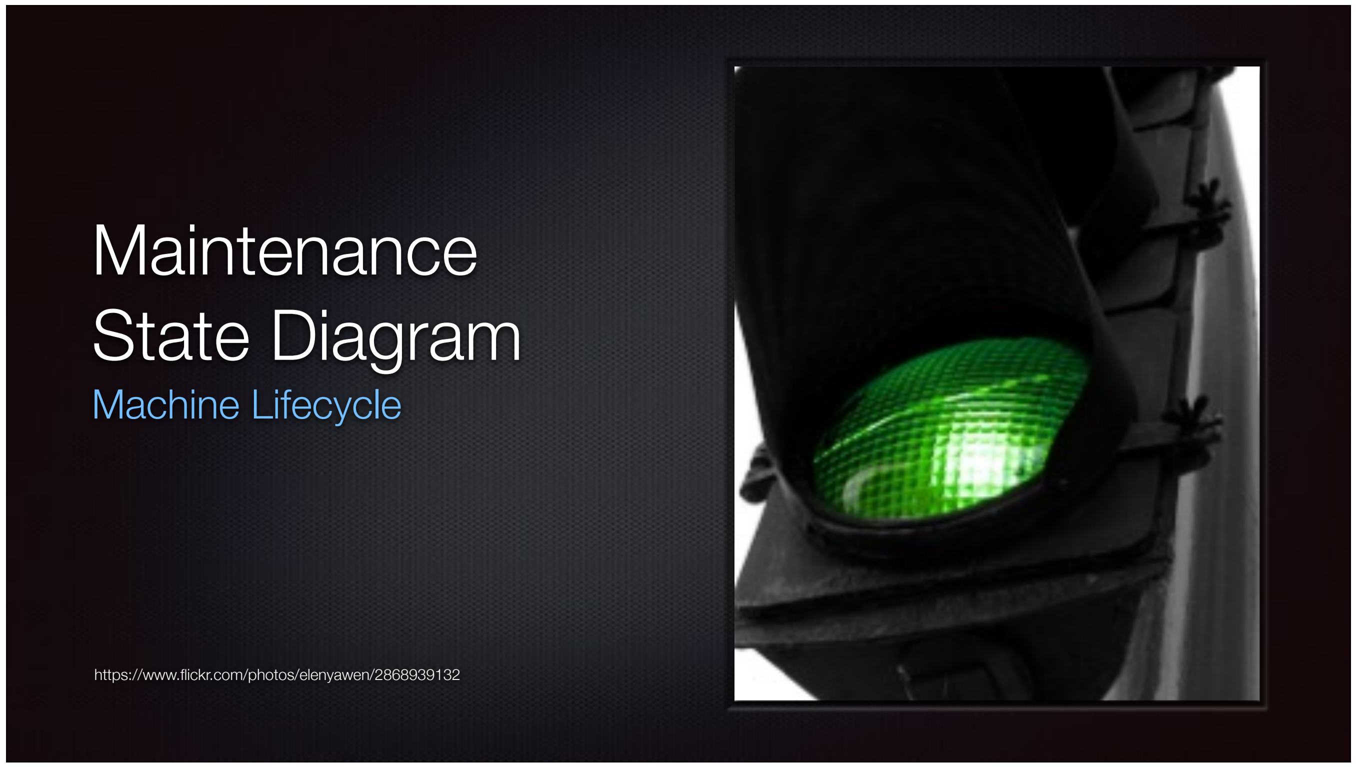
devcluster / www-data / prod / hello

Active tasks (1) | Completed tasks (1) | All tasks

Instance	Status	Running duration	Host
0	− 3 minutes ago - LOST : Slave 192.168.33.7 removed: health check timed out • 10/08 11:49:46 LOCAL • PENDING • 10/08 11:49:46 LOCAL • ASSIGNED • 10/08 11:49:48 LOCAL • STARTING • Initializing sandbox. • 10/08 11:49:48 LOCAL • RUNNING • 10/08 11:51:23 LOCAL • LOST • Slave 192.168.33.7 removed: health check timed out	2 minutes	192.168.33.7

When we were doing maintenance- this is how our users would know. Hundreds of these “completed tasks” gone LOST. We would need to send out huge email messages letting our users know to expect lots of cluster churn, and to silence alerts for flapping instances… since it was all “normal.”

Also, Aurora and Mesos oncalls would be notified that we were losing slaves and tasks- meaning our team-internal communication needed to be flawless.



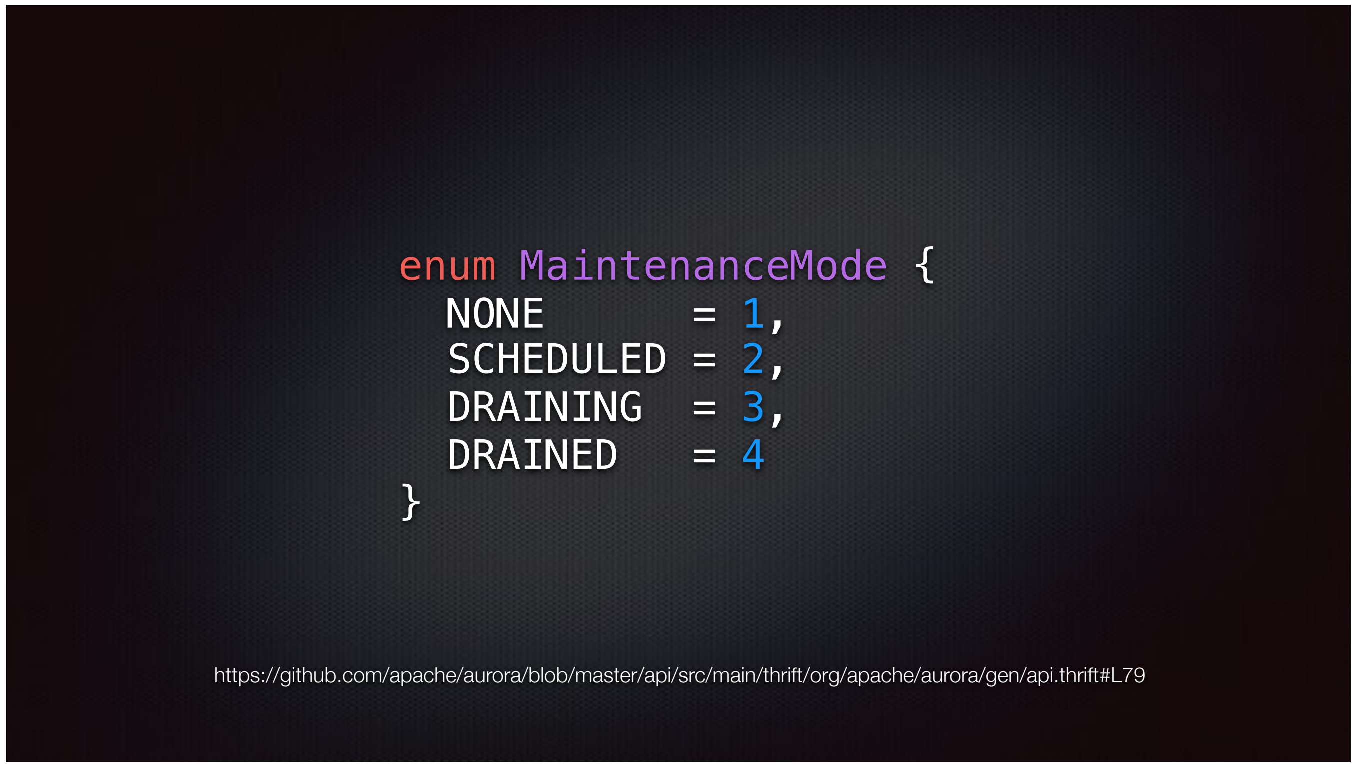
Maintenance State Diagram

Machine Lifecycle

https://www.flickr.com/photos/elenyawen/2868939132

This couldn’t scale. We needed a better way to communicate maintenance, without slowing ourselves down.

We essentially put traffic control on our maintenance- this empowered the stop/go logic we needed to safely traverse our machines

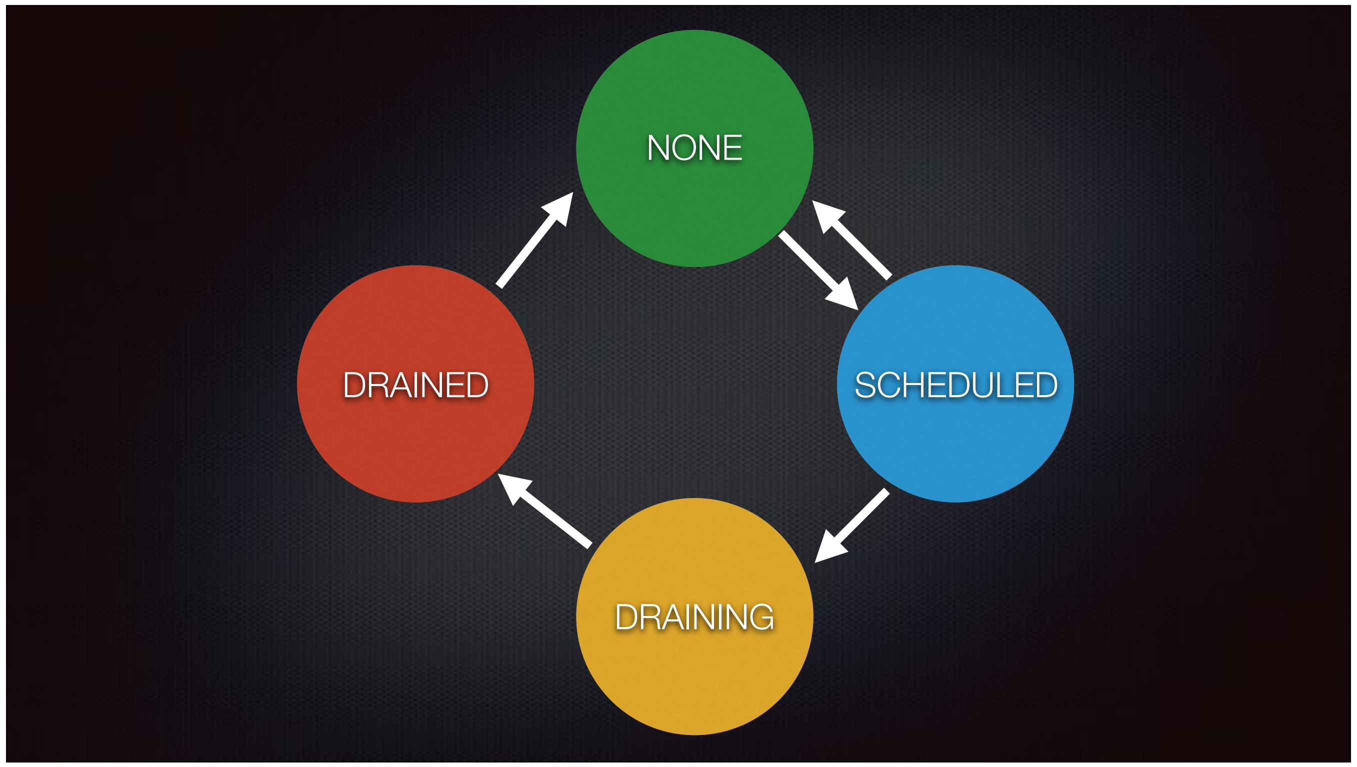


```
enum MaintenanceMode {
  NONE      = 1,
  SCHEDULED = 2,
  DRAINING  = 3,
  DRAINED   = 4
}
```

https://github.com/apache/aurora/blob/master/api/src/main/thrift/org/apache/aurora/gen/api.thrift#L79

Here’s the set of states a machine can be in.

Aurora implements “two-tiered scheduling”

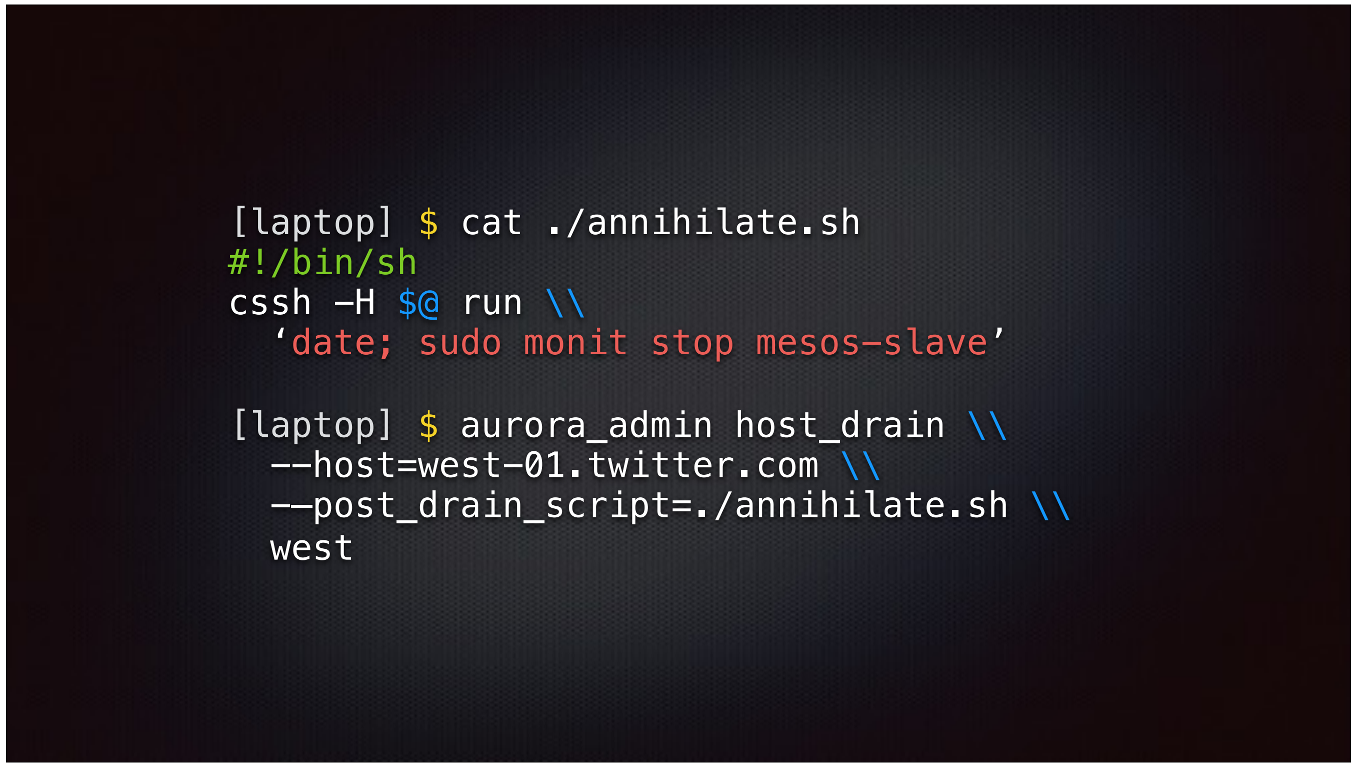


A machine is normally happy- it has no MaintenanceMode.

When we put a large set of hosts into “scheduled”.. it tells Aurora to defer scheduling on those machines, as we’re planning to drain them. This helps avoid tasks playing leapfrog from machine to machine.

When we tell Aurora it’s time to take hosts down, it puts a machine into DRAINING, killing its tasks.

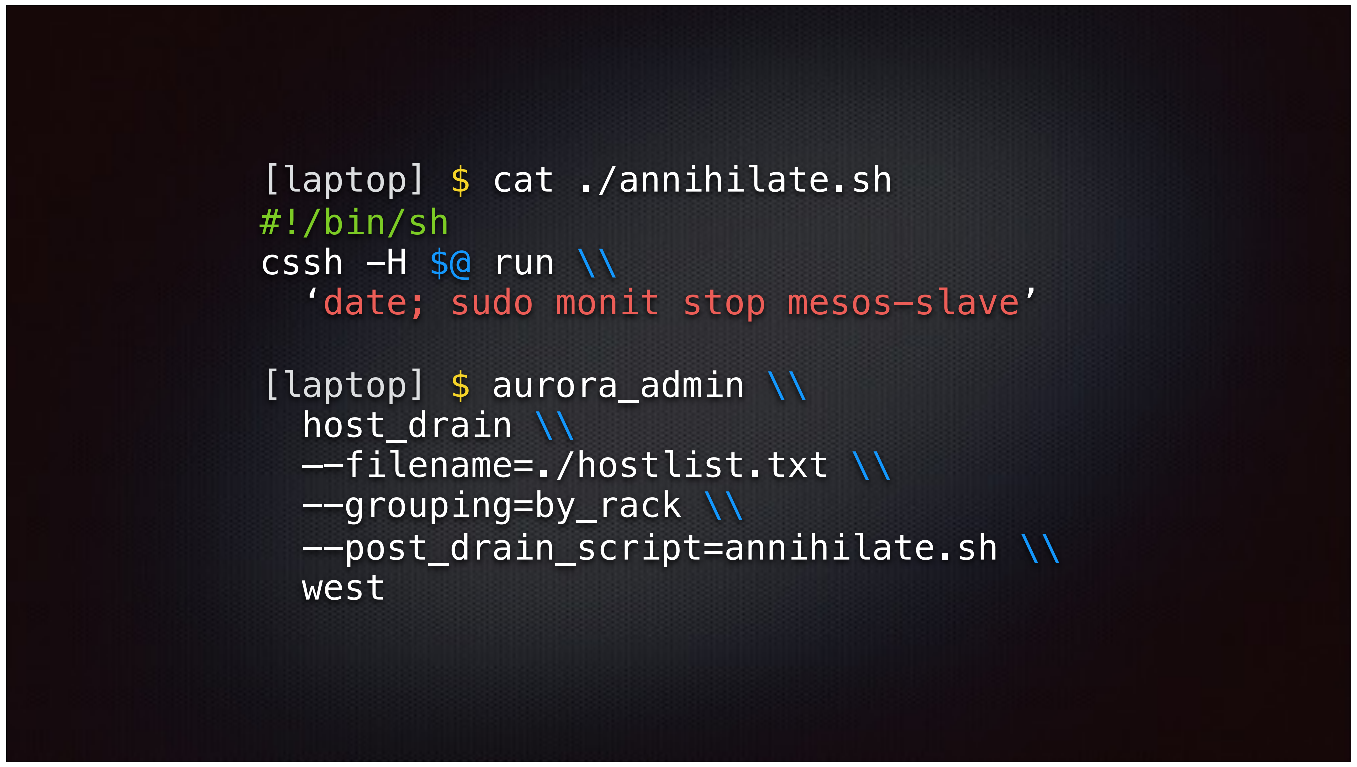
At the end, it will put it into drained when it’s all set.



```
[laptop] $ cat ./annihilate.sh
#!/bin/sh
cssh -H $@ run \\
  ‘date; sudo monit stop mesos-slave’

[laptop] $ aurora_admin host_drain \\
  --host=west-01.twitter.com \\
  --post_drain_script=./annihilate.sh \\
  west
```

How does this look?

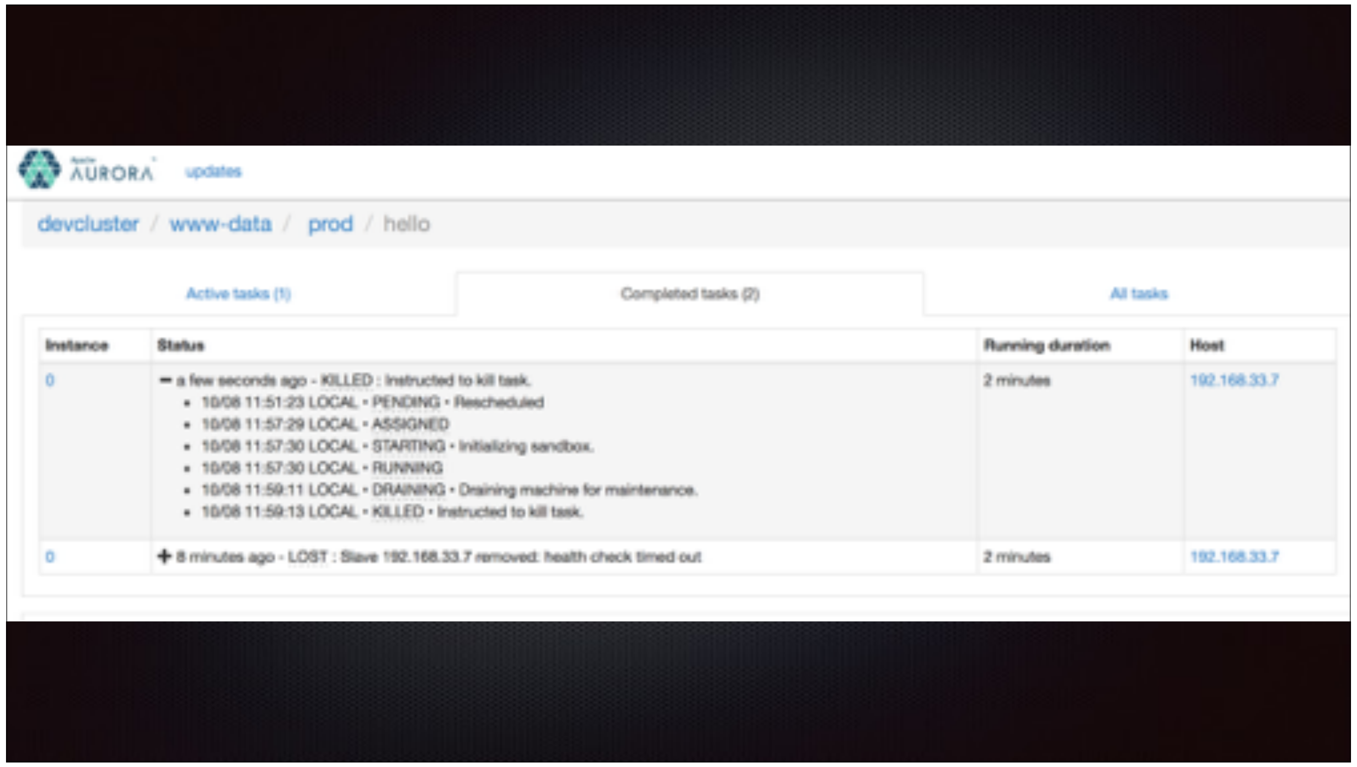
With one host…



```
[laptop] $ cat ./annihilate.sh
#!/bin/sh
cssh -H $@ run \\
  'date; sudo monit stop mesos-slave'

[laptop] $ aurora_admin \\
  host_drain \\
  --filename=./hostlist.txt \\
  --grouping=by_rack \\
  --post_drain_script=annihilate.sh \\
  west
```

We were able to move “quickly” through the cluster without paging ourselves… but instead we would cause issues for our users- their SLAs would be affected since we did not hold ourselves to any standard

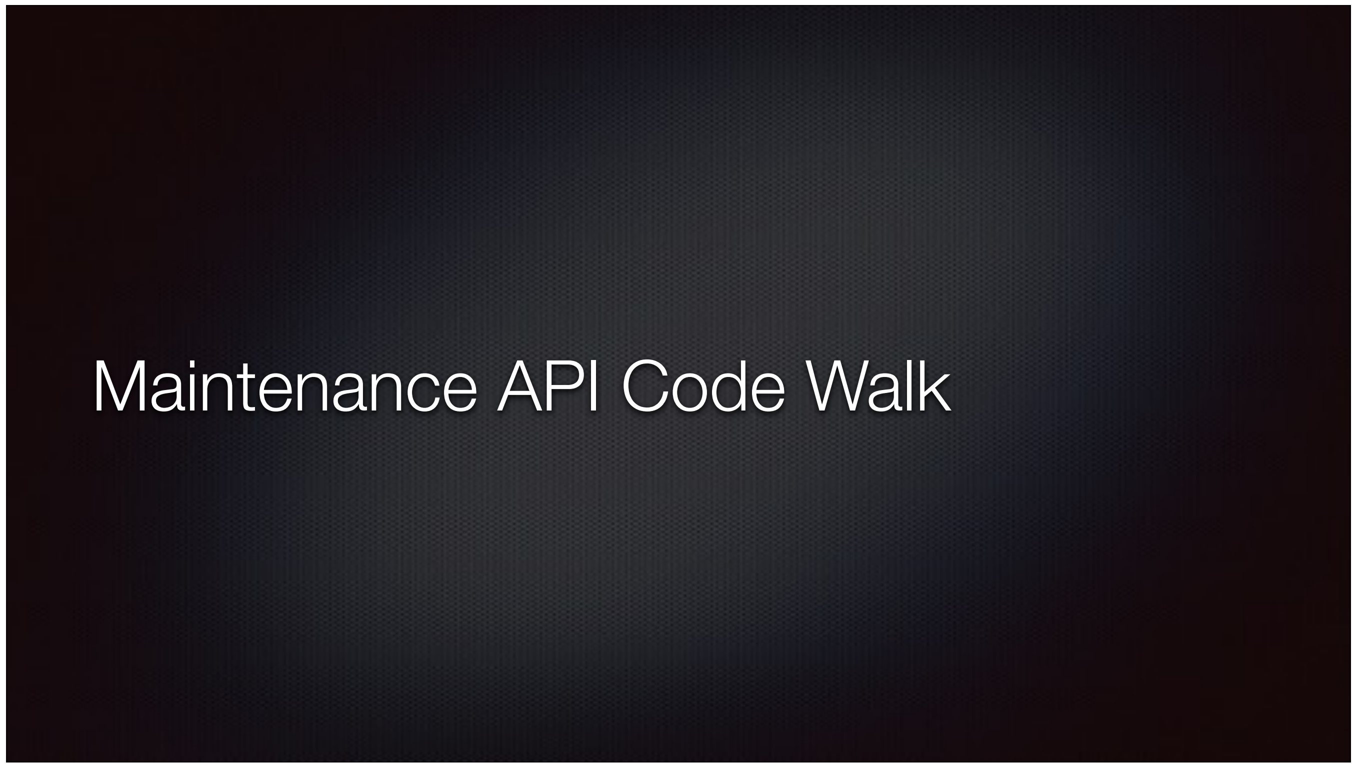
We have a special “grouping” where we will actually form the hosts into sets based on the rack of the machine- this allowed us to only take down one rack at a time- which service owners were already prepared to sustain in case of power/network failure.



devcluster / www-data / prod / hello

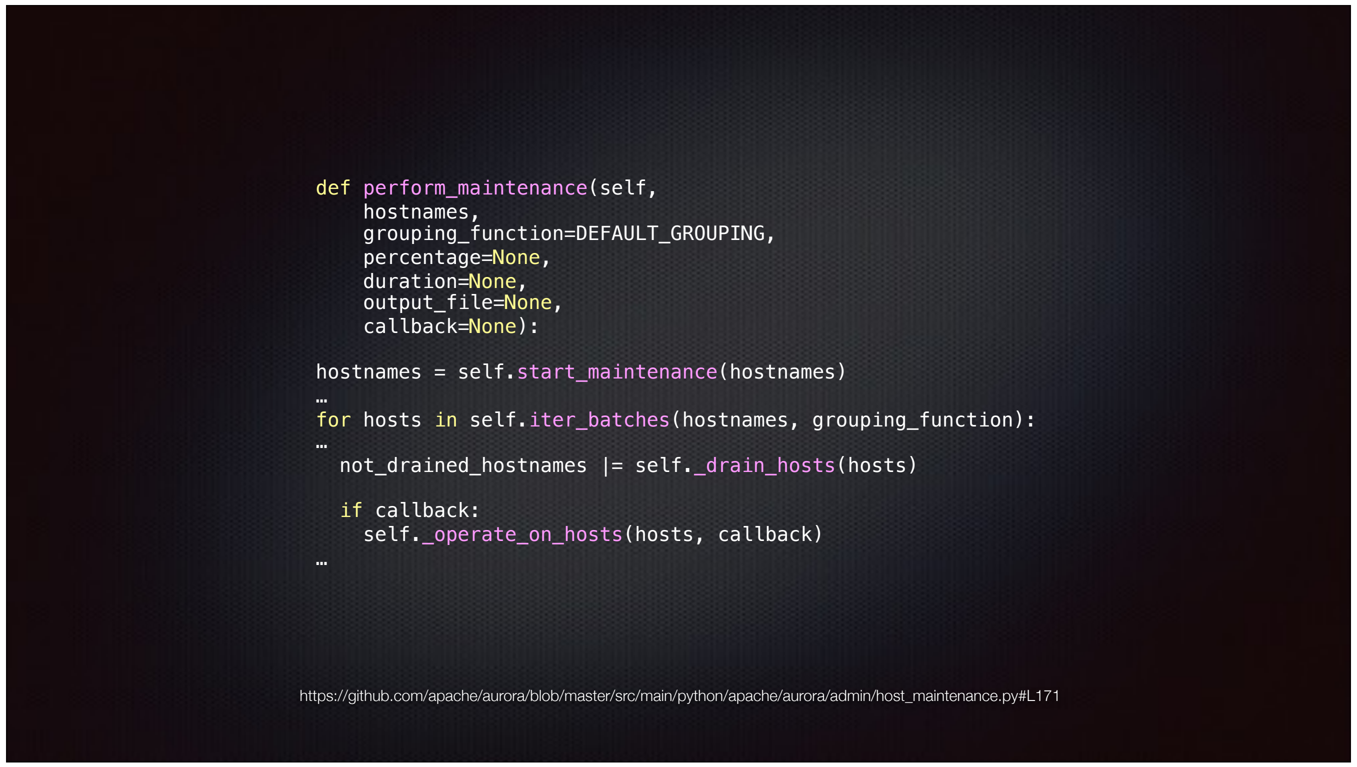
Instance	Status	Running duration	Host
0	− a few seconds ago - KILLED : Instructed to kill task. • 10/08 11:51:23 LOCAL • PENDING • Rescheduled • 10/08 11:57:29 LOCAL • ASSIGNED • 10/08 11:57:30 LOCAL • STARTING • Initializing sandbox. • 10/08 11:57:30 LOCAL • RUNNING • 10/08 11:59:11 LOCAL • DRAINING • Draining machine for maintenance. • 10/08 11:59:13 LOCAL • KILLED • Instructed to kill task.	2 minutes	192.168.33.7
0	+ 8 minutes ago - LOST : Slave 192.168.33.7 removed: health check timed out	2 minutes	192.168.33.7

Now, they got a much better message



Maintenance API Code Walk

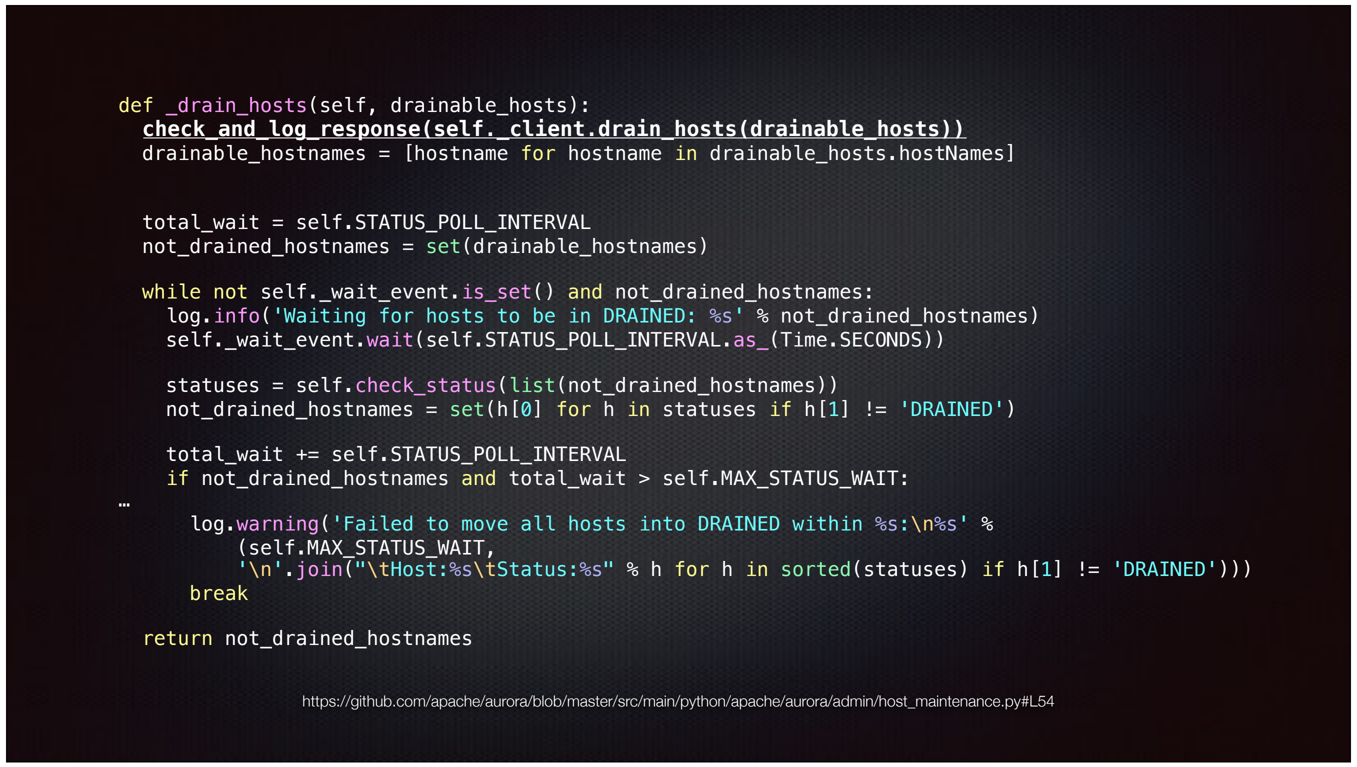
You might need to walk through the actual aurora_admin or scheduler code, so let’s take a look at how this is implemented.



```
def perform_maintenance(self,
    hostnames,
    grouping_function=DEFAULT_GROUPING,
    percentage=None,
    duration=None,
    output_file=None,
    callback=None):

hostnames = self.start_maintenance(hostnames)
…
for hosts in self.iter_batches(hostnames, grouping_function):
…
  not_drained_hostnames |= self._drain_hosts(hosts)

  if callback:
    self._operate_on_hosts(hosts, callback)
…
```

https://github.com/apache/aurora/blob/master/src/main/python/apache/aurora/admin/host_maintenance.py#L171

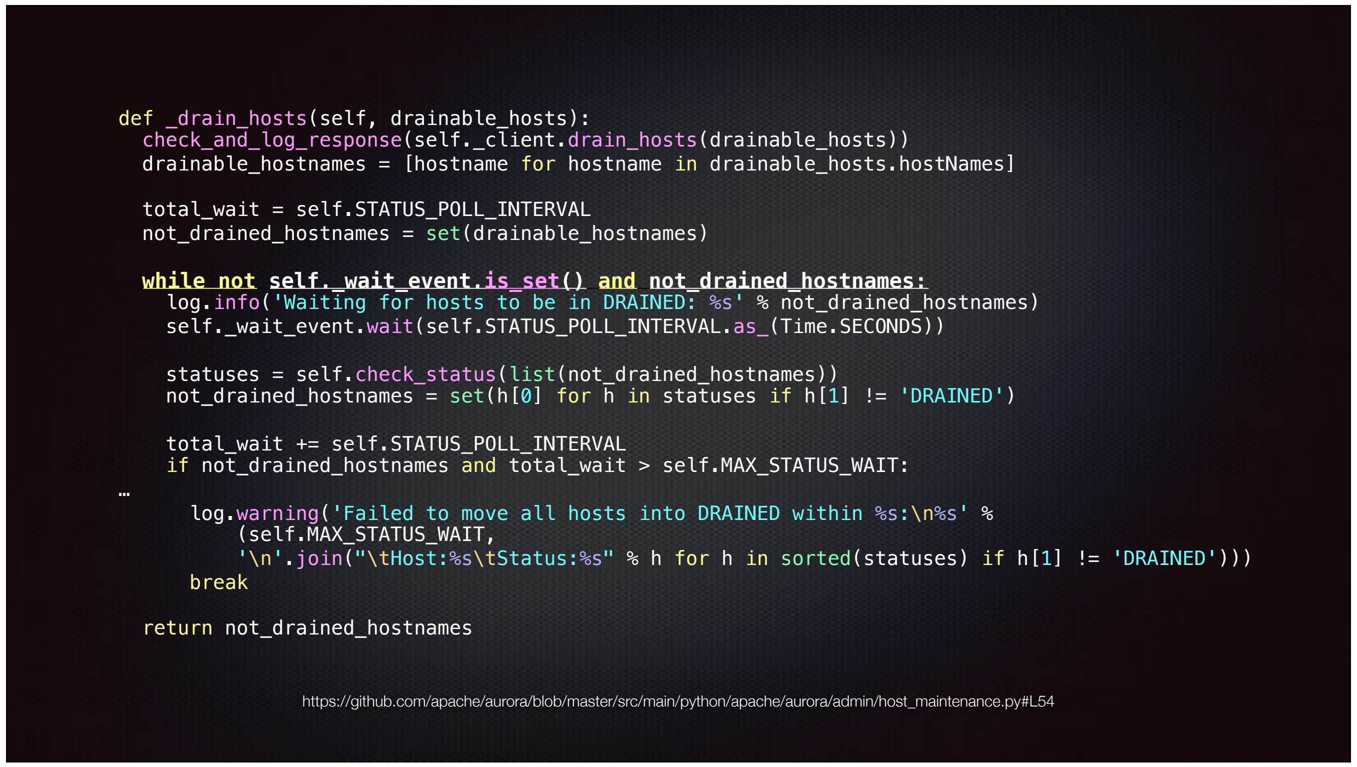
Again, we were able to break it up into “batches” -> we would do it by rack



```
def _drain_hosts(self, drainable_hosts):
  check_and_log_response(self._client.drain_hosts(drainable_hosts))
  drainable_hostnames = [hostname for hostname in drainable_hosts.hostNames]


  total_wait = self.STATUS_POLL_INTERVAL
  not_drained_hostnames = set(drainable_hostnames)

  while not self._wait_event.is_set() and not_drained_hostnames:
    log.info('Waiting for hosts to be in DRAINED: %s' % not_drained_hostnames)
    self._wait_event.wait(self.STATUS_POLL_INTERVAL.as_(Time.SECONDS))

    statuses = self.check_status(list(not_drained_hostnames))
    not_drained_hostnames = set(h[0] for h in statuses if h[1] != 'DRAINED')

    total_wait += self.STATUS_POLL_INTERVAL
    if not_drained_hostnames and total_wait > self.MAX_STATUS_WAIT:
…
      log.warning('Failed to move all hosts into DRAINED within %s:\n%s' %
          (self.MAX_STATUS_WAIT,
          '\n'.join("\tHost:%s\tStatus:%s" % h for h in sorted(statuses) if h[1] != 'DRAINED')))
      break

  return not_drained_hostnames
```

https://github.com/apache/aurora/blob/master/src/main/python/apache/aurora/admin/host_maintenance.py#L54

Here's where we actually drain the hosts- we're using the Aurora client API to send an RPC to the scheduler



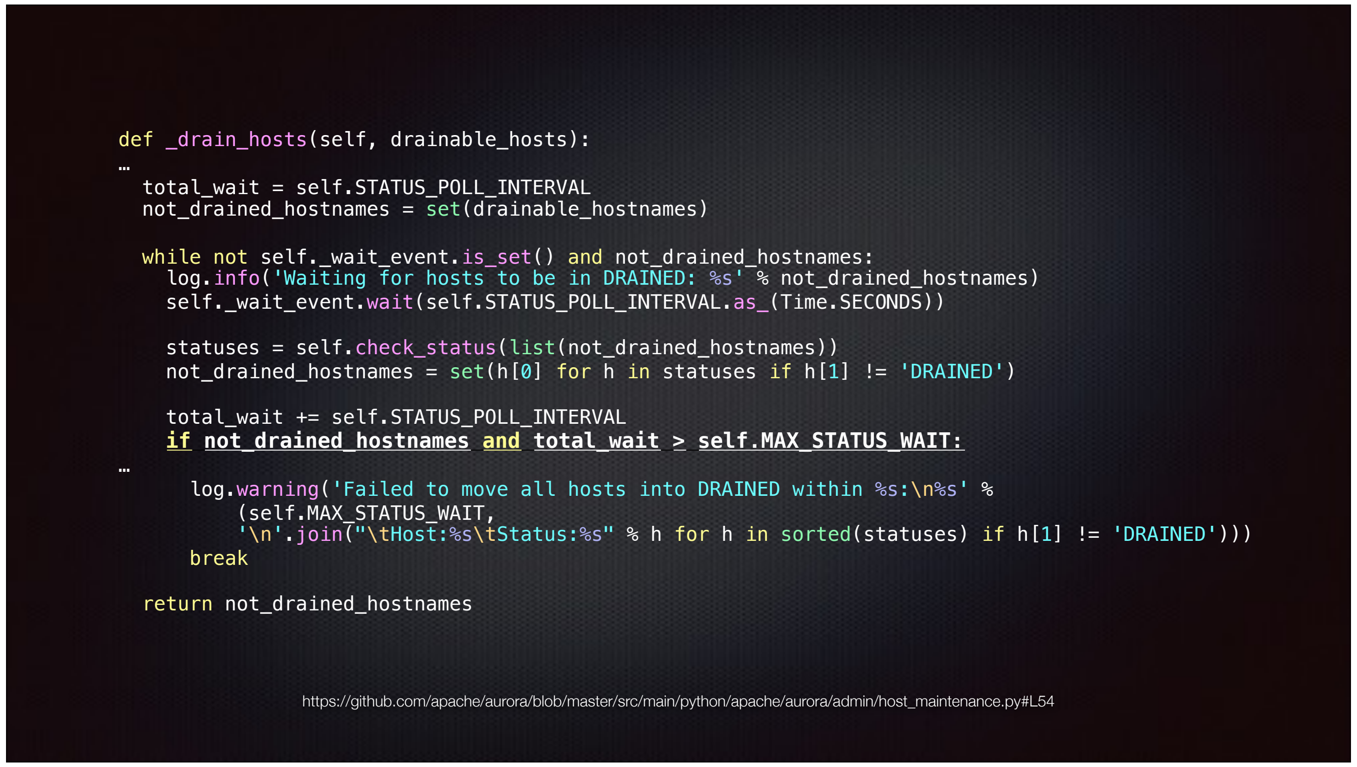
```
def _drain_hosts(self, drainable_hosts):
  check_and_log_response(self._client.drain_hosts(drainable_hosts))
  drainable_hostnames = [hostname for hostname in drainable_hosts.hostNames]

  total_wait = self.STATUS_POLL_INTERVAL
  not_drained_hostnames = set(drainable_hostnames)

  while not self._wait_event.is_set() and not_drained_hostnames:
    log.info('Waiting for hosts to be in DRAINED: %s' % not_drained_hostnames)
    self._wait_event.wait(self.STATUS_POLL_INTERVAL.as_(Time.SECONDS))

    statuses = self.check_status(list(not_drained_hostnames))
    not_drained_hostnames = set(h[0] for h in statuses if h[1] != 'DRAINED')

    total_wait += self.STATUS_POLL_INTERVAL
    if not_drained_hostnames and total_wait > self.MAX_STATUS_WAIT:
…
      log.warning('Failed to move all hosts into DRAINED within %s:\n%s' %
          (self.MAX_STATUS_WAIT,
          '\n'.join("\tHost:%s\tStatus:%s" % h for h in sorted(statuses) if h[1] != 'DRAINED')))
      break

  return not_drained_hostnames
```

https://github.com/apache/aurora/blob/master/src/main/python/apache/aurora/admin/host_maintenance.py#L54

At this point, we're going to poll the scheduler for a certain timeout to make sure these hosts are drained of user tasks

```
def _drain_hosts(self, drainable_hosts):
…
  total_wait = self.STATUS_POLL_INTERVAL
  not_drained_hostnames = set(drainable_hostnames)

  while not self._wait_event.is_set() and not_drained_hostnames:
    log.info('Waiting for hosts to be in DRAINED: %s' % not_drained_hostnames)
    self._wait_event.wait(self.STATUS_POLL_INTERVAL.as_(Time.SECONDS))

    statuses = self.check_status(list(not_drained_hostnames))
    not_drained_hostnames = set(h[0] for h in statuses if h[1] != 'DRAINED')

    total_wait += self.STATUS_POLL_INTERVAL
    if not_drained_hostnames and total_wait > self.MAX_STATUS_WAIT:
…
      log.warning('Failed to move all hosts into DRAINED within %s:\n%s' %
          (self.MAX_STATUS_WAIT,
          '\n'.join("\tHost:%s\tStatus:%s" % h for h in sorted(statuses) if h[1] != 'DRAINED')))
      break

  return not_drained_hostnames
```

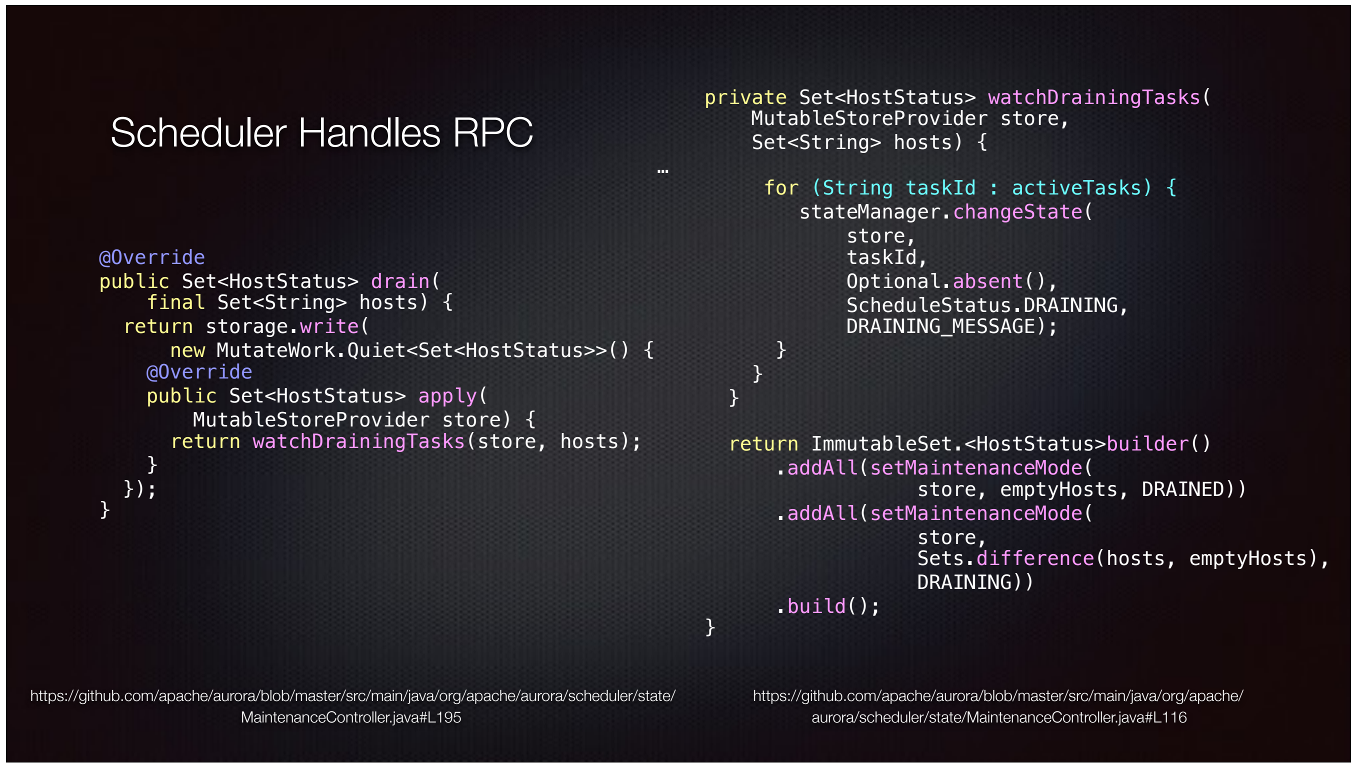
https://github.com/apache/aurora/blob/master/src/main/python/apache/aurora/admin/host_maintenance.py#L54



```
def _drain_hosts(self, drainable_hosts):
…
  total_wait = self.STATUS_POLL_INTERVAL
  not_drained_hostnames = set(drainable_hostnames)

  while not self._wait_event.is_set() and not_drained_hostnames:
    log.info('Waiting for hosts to be in DRAINED: %s' % not_drained_hostnames)
    self._wait_event.wait(self.STATUS_POLL_INTERVAL.as_(Time.SECONDS))

    statuses = self.check_status(list(not_drained_hostnames))
    not_drained_hostnames = set(h[0] for h in statuses if h[1] != 'DRAINED')

    total_wait += self.STATUS_POLL_INTERVAL
    if not_drained_hostnames and total_wait > self.MAX_STATUS_WAIT:
…
      log.warning('Failed to move all hosts into DRAINED within %s:\n%s' %
          (self.MAX_STATUS_WAIT,
          '\n'.join("\tHost:%s\tStatus:%s" % h for h in sorted(statuses) if h[1] != 'DRAINED')))
      break

  return not_drained_hostnames
```

https://github.com/apache/aurora/blob/master/src/main/python/apache/aurora/admin/host_maintenance.py#L54

And finally we'll time out



Scheduler Handles RPC

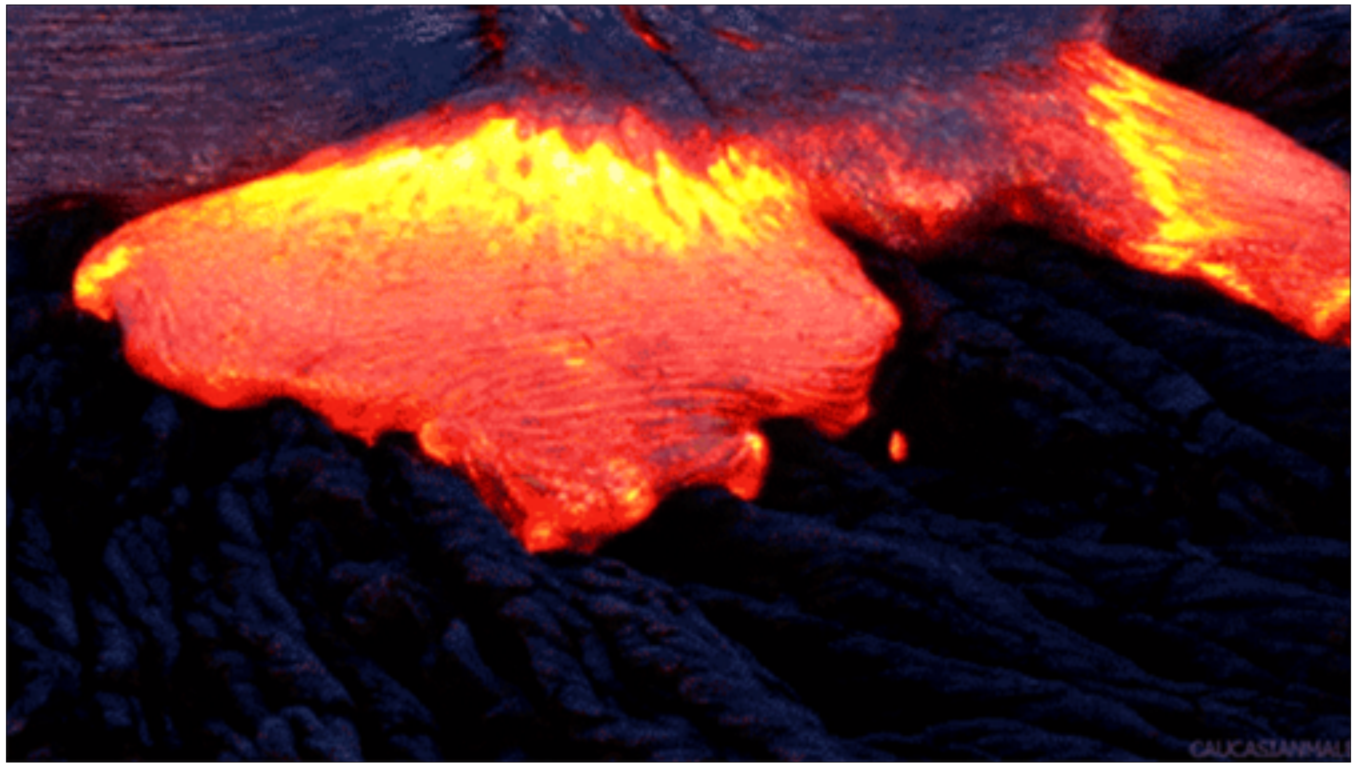
```
@Override
public Set<HostStatus> drain(
    final Set<String> hosts) {
  return storage.write(
      new MutateWork.Quiet<Set<HostStatus>>() {
    @Override
    public Set<HostStatus> apply(
        MutableStoreProvider store) {
      return watchDrainingTasks(store, hosts);
    }
  });
}
```

https://github.com/apache/aurora/blob/master/src/main/java/org/apache/aurora/scheduler/state/MaintenanceController.java#L195

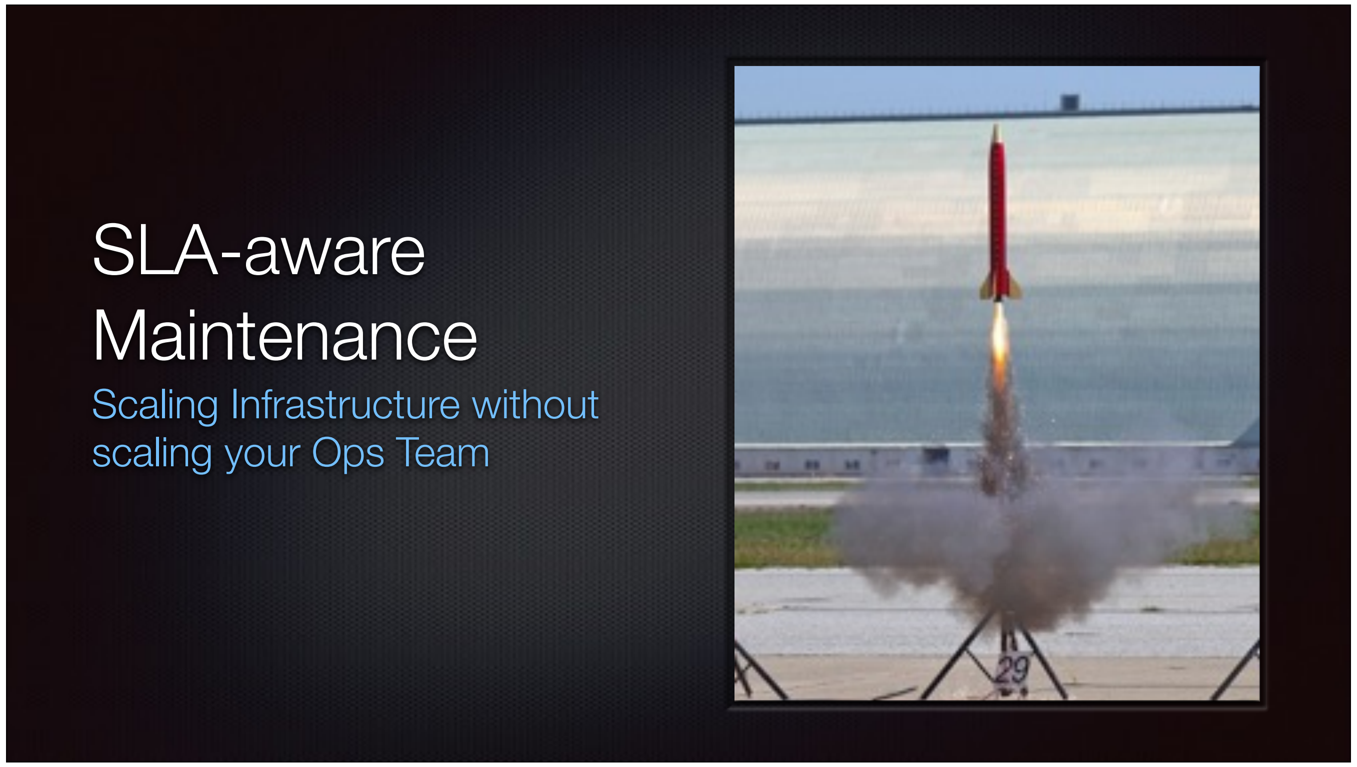
```
private Set<HostStatus> watchDrainingTasks(
    MutableStoreProvider store,
    Set<String> hosts) {
…
     for (String taskId : activeTasks) {
        stateManager.changeState(
            store,
            taskId,
            Optional.absent(),
            ScheduleStatus.DRAINING,
            DRAINING_MESSAGE);
      }
    }
  }

  return ImmutableSet.<HostStatus>builder()
      .addAll(setMaintenanceMode(
                  store, emptyHosts, DRAINED))
      .addAll(setMaintenanceMode(
                  store,
                  Sets.difference(hosts, emptyHosts),
                  DRAINING))
      .build();
}
```

https://github.com/apache/aurora/blob/master/src/main/java/org/apache/aurora/scheduler/state/MaintenanceController.java#L116

Then the maintenance controller will get called with its taskChangedState when any of those DRAINING tasks get called



After we got the initial tooling, things actually went a bit like this- the lava went all over the place?



SLA-aware Maintenance

Scaling Infrastructure without scaling your Ops Team

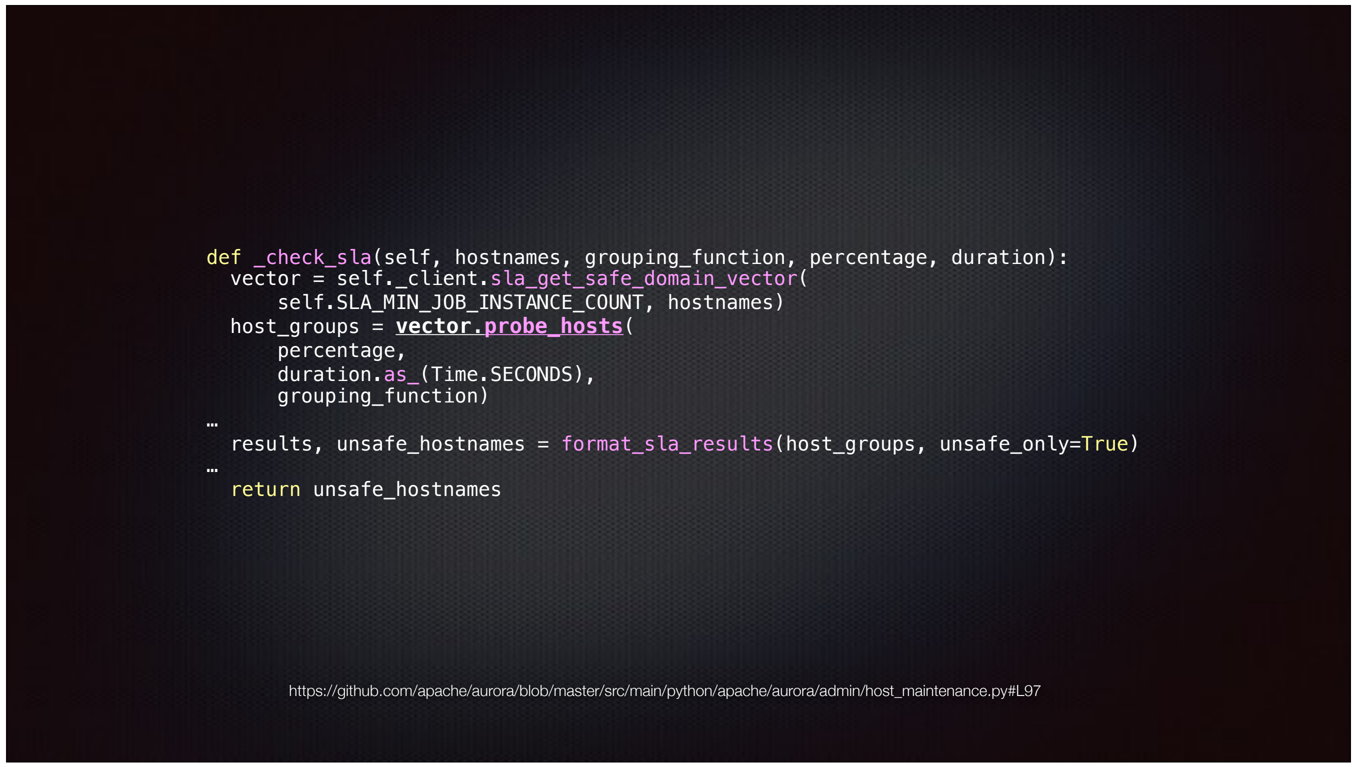
We needed to add some controls- slow things down and cause a controlled explosion

```python
def perform_maintenance(self,
    hostnames,
  grouping_function=DEFAULT_GROUPING,
  percentage=None,
  duration=None,
  output_file=None,
  callback=None):
…
for hosts in self.iter_batches(hostnames, grouping_function):
  log.info('Beginning SLA check for %s' % hosts.hostNames)
  unsafe_hostnames = self._check_sla(
      list(hosts.hostNames),
      grouping_function,
      percentage,
      duration)
  if unsafe_hostnames:
    log.warning('Some hosts did not pass SLA check and will '
                'not be drained! '
                'Skipping hosts: %s' % unsafe_hostnames)
…
  if callback:
    self._operate_on_hosts(hosts, callback)
```

https://github.com/apache/aurora/blob/master/src/main/python/apache/aurora/admin/host_maintenance.py#L171

```
def _check_sla(self, hostnames, grouping_function, percentage, duration):
  vector = self._client.sla_get_safe_domain_vector(
      self.SLA_MIN_JOB_INSTANCE_COUNT, hostnames)
  host_groups = vector.probe_hosts(
      percentage,
      duration.as_(Time.SECONDS),
      grouping_function)
…
  results, unsafe_hostnames = format_sla_results(host_groups, unsafe_only=True)
…
  return unsafe_hostnames
```

https://github.com/apache/aurora/blob/master/src/main/python/apache/aurora/admin/host_maintenance.py#L97



```
def _check_sla(self, hostnames, grouping_function, percentage, duration):
  vector = self._client.sla_get_safe_domain_vector(
      self.SLA_MIN_JOB_INSTANCE_COUNT, hostnames)
  host_groups = vector.probe_hosts(
      percentage,
      duration.as_(Time.SECONDS),
      grouping_function)
…
  results, unsafe_hostnames = format_sla_results(host_groups, unsafe_only=True)
…
  return unsafe_hostnames
```

https://github.com/apache/aurora/blob/master/src/main/python/apache/aurora/admin/host_maintenance.py#L97

Let’s look into this SLA Vector and how it checks the SLA of tasks on the host

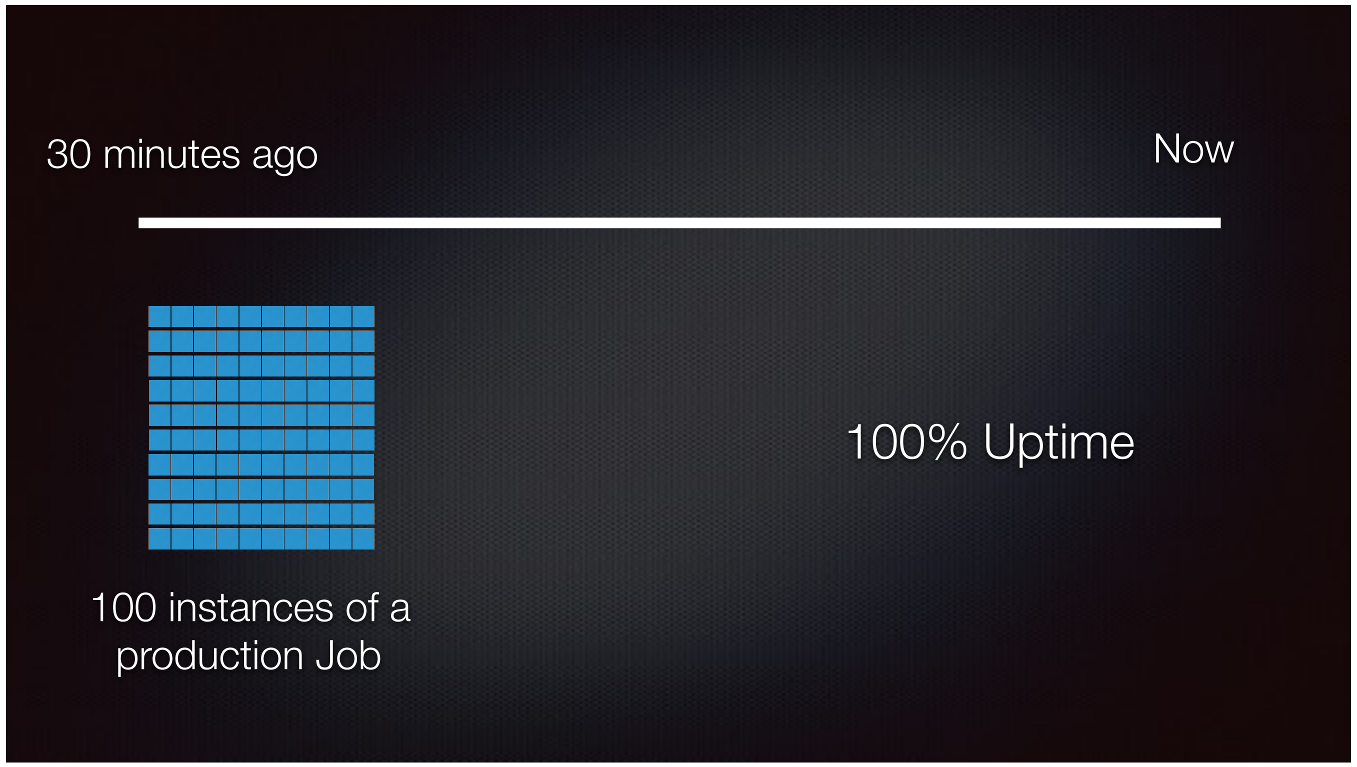
```python
def probe_hosts(self, percentage, duration, grouping_function=DEFAULT_GROUPING):
…
  for job_key in job_keys:
    job_hosts = hosts.intersection(self._hosts_by_job[job_key])
    filtered_percentage, total_count, filtered_vector = self._simulate_hosts_down(
        job_key, job_hosts, duration)

    # Calculate wait time to SLA in case down host violates job's SLA.
    if filtered_percentage < percentage:
      safe = False
      wait_to_sla = filtered_vector.get_wait_time_to_sla(percentage, duration, total_count)
    else:
      safe = True
      wait_to_sla = 0
…
```

https://github.com/apache/aurora/blob/master/src/main/python/apache/aurora/client/api/sla.py#L199

```python
def _simulate_hosts_down(self, job_key, hosts, duration):
  unfiltered_tasks = self._tasks_by_job[job_key]

  # Get total job task count to use in SLA calculation.
  total_count = len(unfiltered_tasks)

  # Get a list of job tasks that would remain after the affected hosts go down
  # and create an SLA vector with these tasks.
  filtered_tasks = [task for task in unfiltered_tasks
                    if task.assignedTask.slaveHost not in hosts]
  filtered_vector = JobUpTimeSlaVector(filtered_tasks, self._now)

  # Calculate the SLA that would be in effect should the host go down.
  filtered_percentage = filtered_vector.get_task_up_count(duration, total_count)
  return filtered_percentage, total_count, filtered_vector
```

https://github.com/apache/aurora/blob/master/src/main/python/apache/aurora/client/api/sla.py#L252

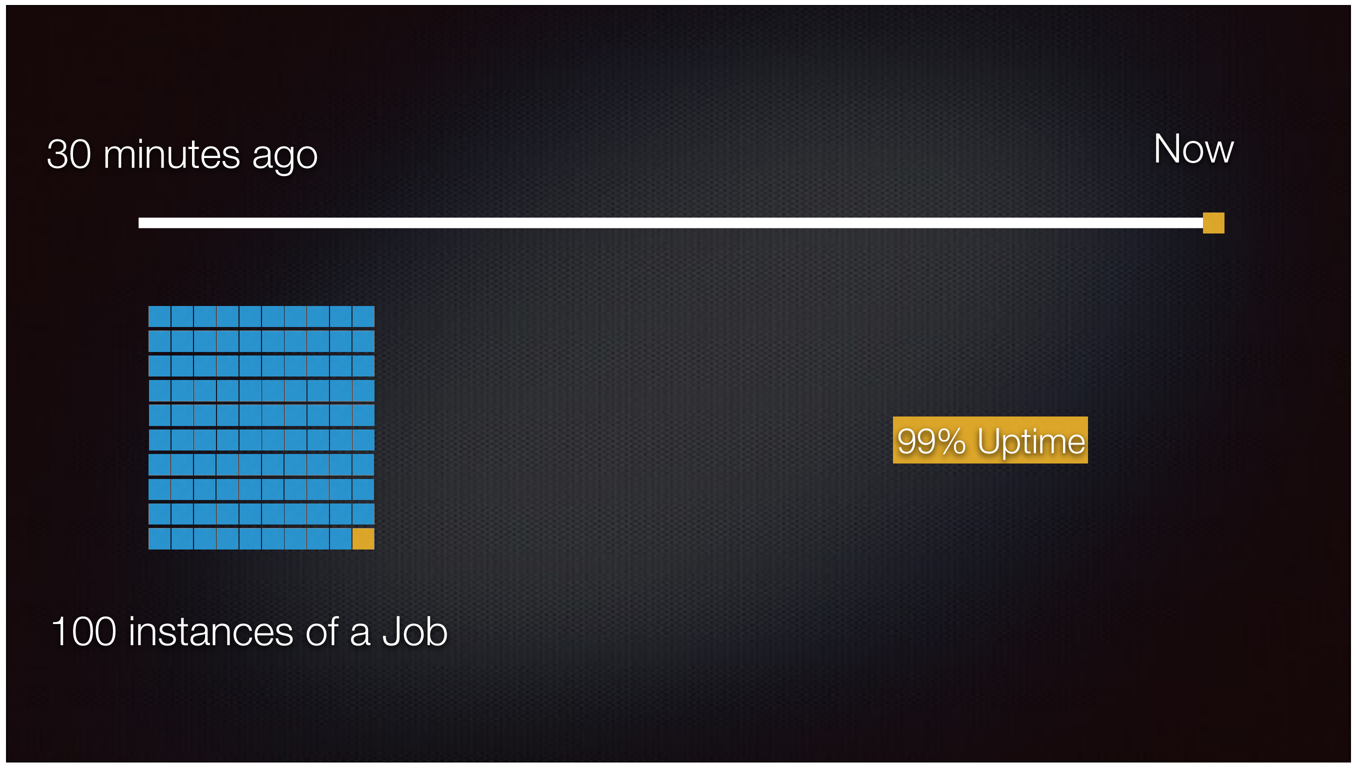
Guarantees 95% over 30 minute SLA



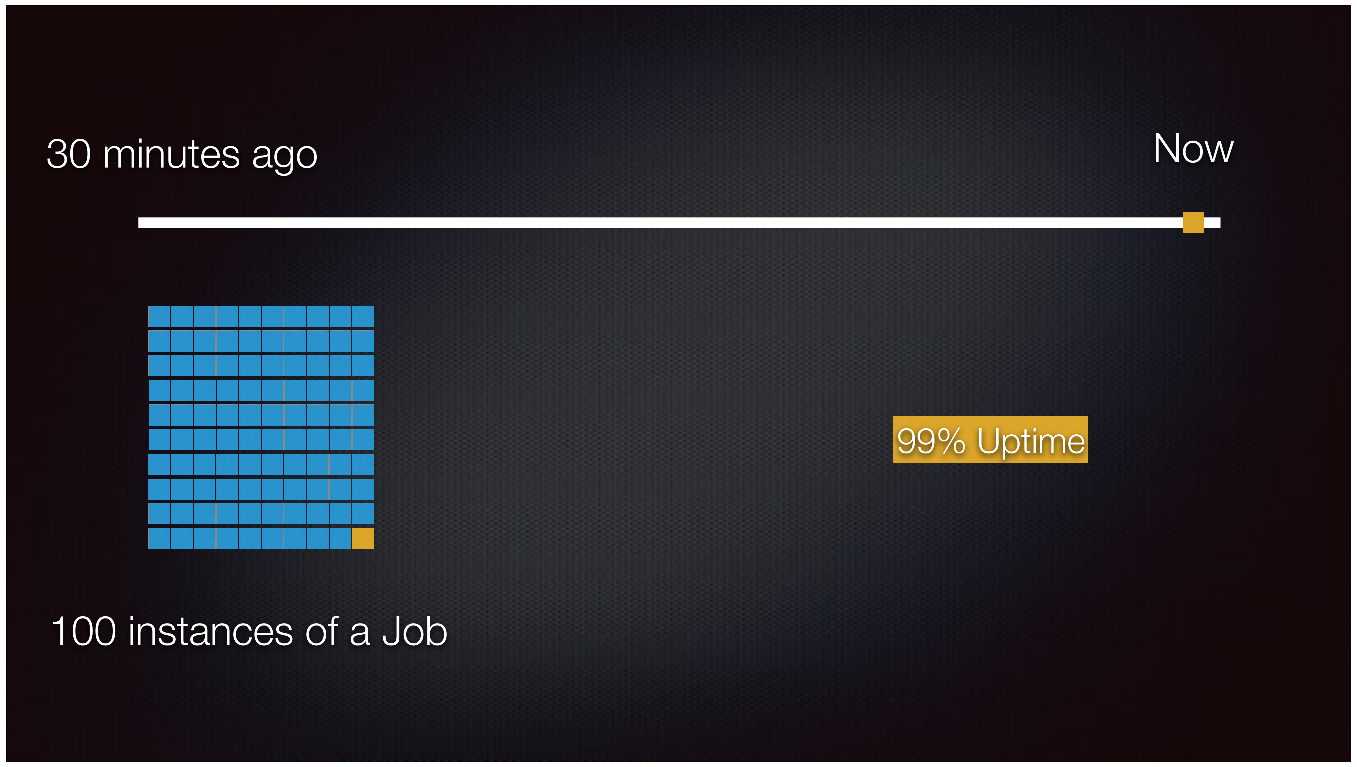
We have a job of 100 instances

Now here’s a timeline- this shows the past 30 minutes. No instances have been restarted, they have all stayed in RUNNING

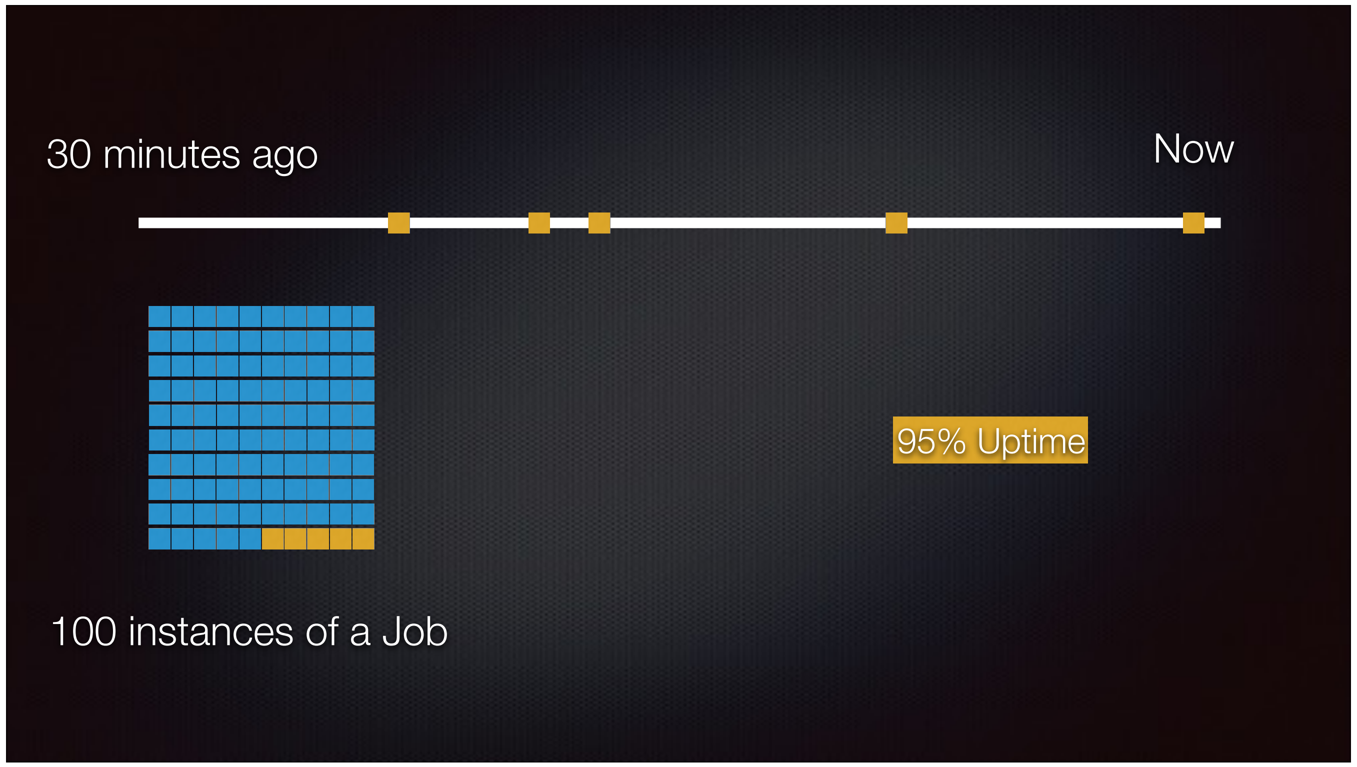
This gives us 100% uptime over 30 minutes



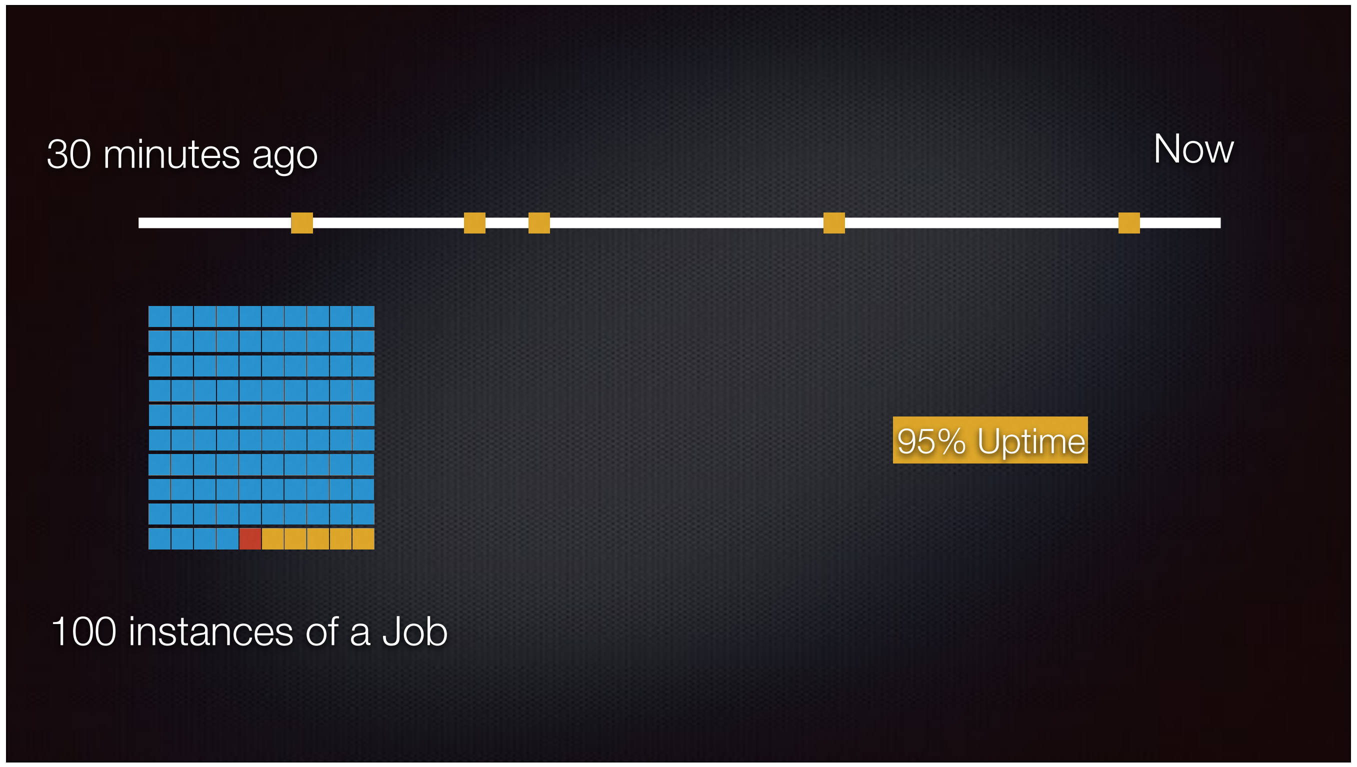
So if we perform maintenance on one of the hosts these instances are running on, we’ll KILL the task… which takes our uptime down to 99%.



And reschedule it somewhere else.

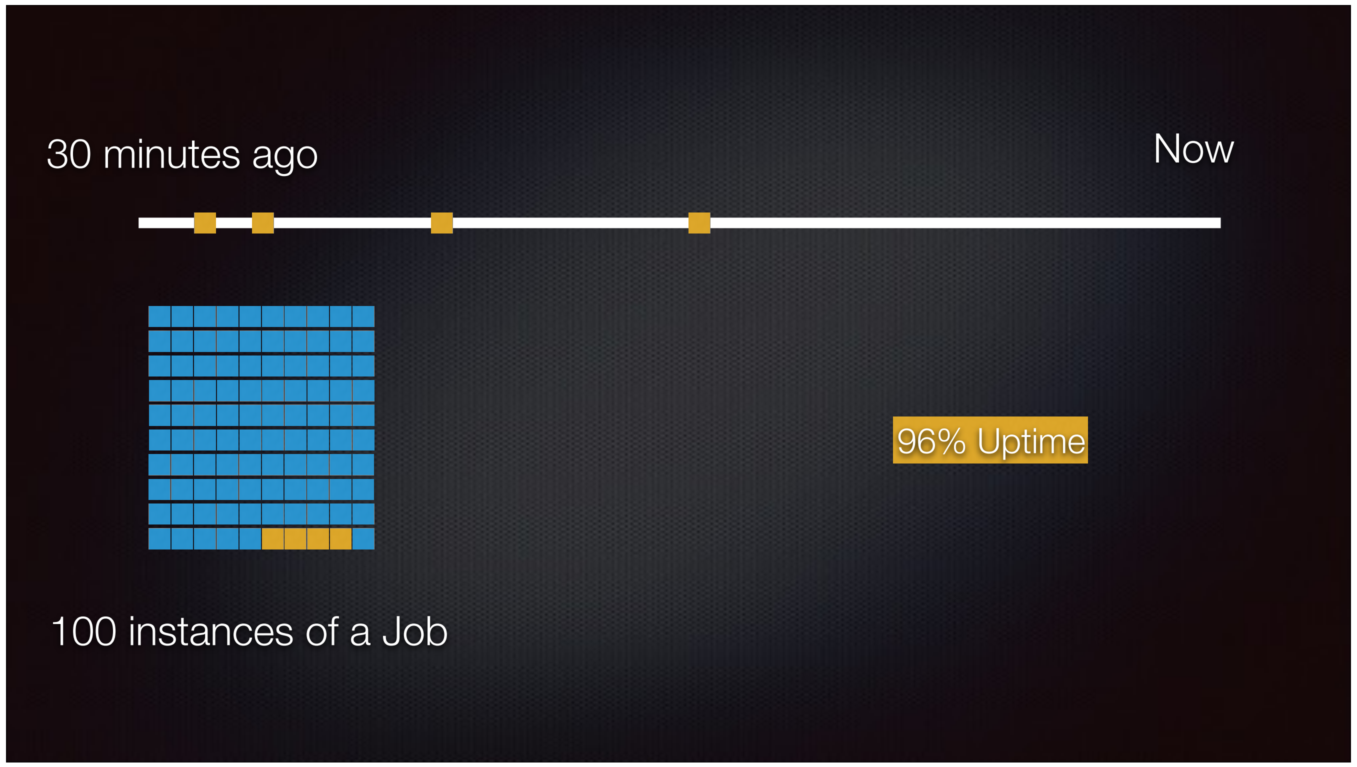


So if we continue with this process across the cluster, we’ll continue taking down instances, until we have killed 5 of these



If we were to probe another host to see if we could take it down- we'd find we could not.

This would take the job's uptime to less than the desired SLA of 95% over 30 minutes. We'll have to wait.

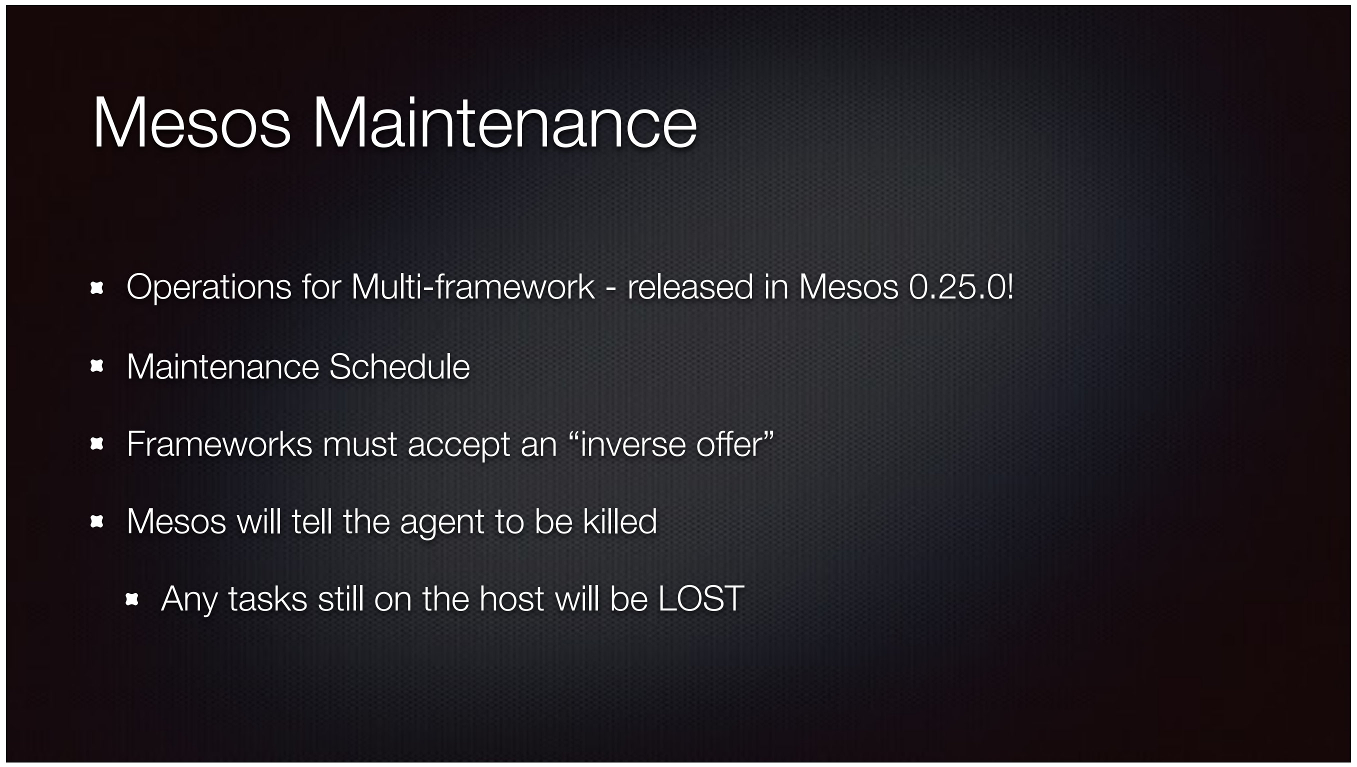


As the time window moves along, we’ll eventually keep an instance alive for longer than 30 minutes, bringing us up to 96% uptime, which means we can take down another instance

The analogy I'd use here is that of a garden- some plants— hosts.. you can prune, others are not ready yet and you need to give them some more time.

Upcoming Features

- Aurora Custom SLA
- Mesos Maintenance Primitives



Mesos Maintenance

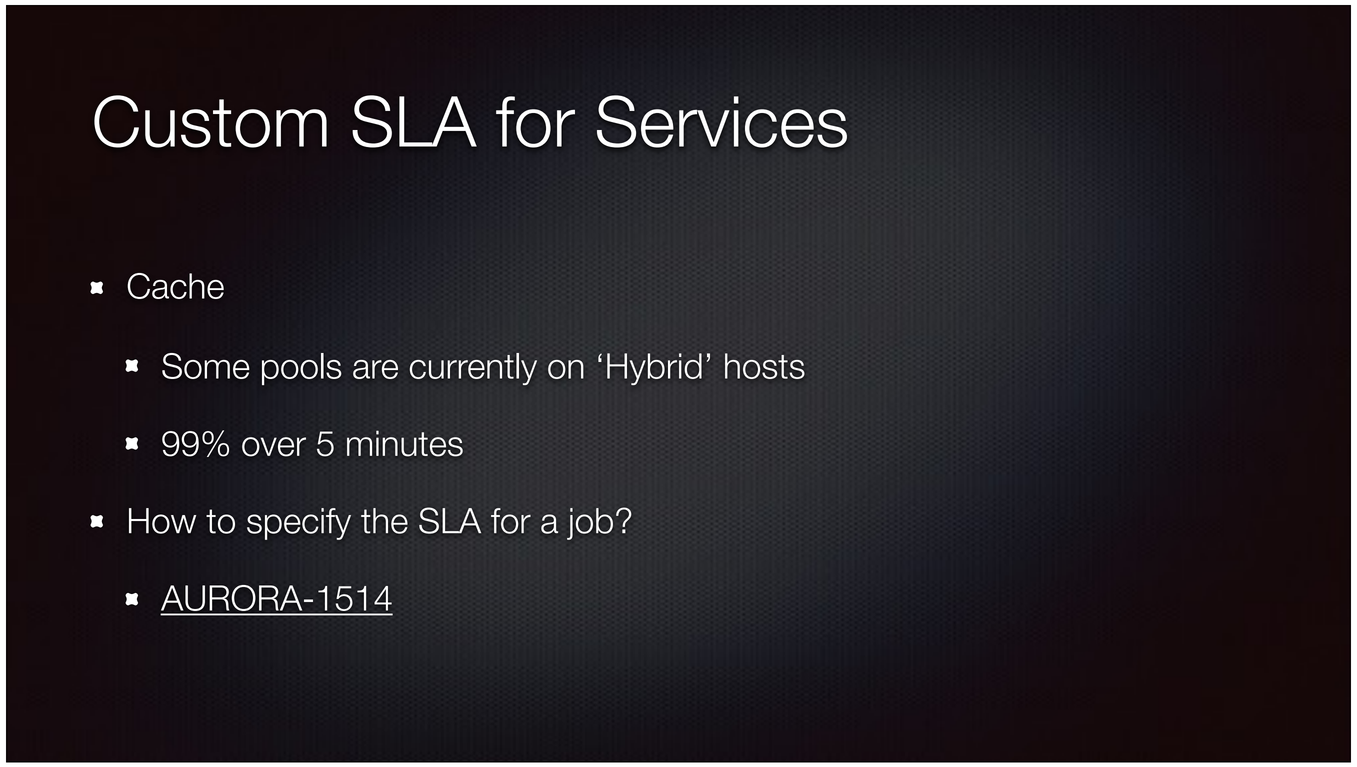
- Operations for Multi-framework - released in Mesos 0.25.0!
- Maintenance Schedule
- Frameworks must accept an “inverse offer”
- Mesos will tell the agent to be killed
  - Any tasks still on the host will be LOST

Most interesting is the ‘maintenance schedule’ - a series of timestamps, each with a set of hosts which are being operated on

The master and operator should perceive acceptance as a best-effort promise by the framework to free all the resources contained in the inverse offer by the start of the unavailability interval.

An inverse offer may also be rejected if the framework is unable to conform to the maintenance schedule.

Frameworks can perform their own scheduling in a maintenance-aware fashion



Custom SLA for Services

- Cache
  - Some pools are currently on ‘Hybrid’ hosts
  - 99% over 5 minutes
- How to specify the SLA for a job?
  - AURORA-1514

This is a component which adds additional complexity- but we’re at the point where there are several use cases which can take advantage of this.

For example.. we have a large fleet of memcache machines. Thanks to the awesome Resource Isolation in Mesos… cache is able to move into our shared cluster without noticeable impact.

Thanks!

- @Yasumoto
- @ApacheAurora, #aurora
- @ApacheMesos, #mesos