# Salt - Conductor Runner Parallelization



## Overview

The Conductor is a Saltstack runner module. It's actually a salt runner module that loads sub modules dynamically based on parameters to the command line interface. It is an orchestration tool build around Saltstack and backend cloud providers. AWS is the only currently supported backend cloud provider.

The tool is a automation interface to a process that wraps the entire end to end delivery, management or removal of software products and services. It's interface is a command line that can be exposed to the desired invocation method such as Rundeck server, Jenkins server or other. It is data driven and uses Saltstack Pillar configuration data to drive the automation with some command line parameters to allow users to create, update, destroy single instances, clusters or full systems in a single command logical way using the Saltstack engine and components of the ecosystem.

With the power that comes with single command end to end provisioning that is both data driven and highly dynamic, comes the need to both control things so as to not stress systems (including aws) as well as to share this capability with many teams/product groups within the organization, or even multiple organizations if designed for that.

To help in both controlling automation and providing shared non-blocking automation between teams/groups, parallel execution of Conductor runtime was added.

This documentation serves to explain the implementation and pieces of the tool that needed to be instance runtime specific.

## Disclaimer

Detailed information about Saltstack itself is out of scope for this document. Some knowledge of Saltstack and it's terminology may be needed for this document to be fully understood.

## Detailed look at the implementation

There are a few areas in the code that needed to be refactored to make Conductor capable of seamless parallelized invocations. These are outline here with some basic information.

### *Conductor Class instance*

A new instance of class Conductor is instantiated with each execution of the runner. All objects and data associated with each run is encapsulated within the Conductor Class or a class instantiated by the Conductor class.

Code for class located here

### *Dynamically generated cloud configuration files*

One of the feature functions of Conductor is that it auto-generates all the required cloud configuration (aws specific data) on each runtime execution. No configuration files are re-used, all are dynamically created from pillar and do not need to be preserved. However, we do store these are potentially will use to archive.

The conductor uses pillar data together with command line parameters to determine what is needs to do when cloud provisioning action is requested and generates the appropriate cloud configuration files.

Three files are generated and used together to provide salt-cloud all that it needs when starting the cloud provisioning in aws via the salt-client running on the conductor. *These salt-cloud and salt-client specific details are out of scope for this document.*

**About CPID (cloud provision id) salt grain**
For the Conductor to be able to archive all dynamic files, logs etc... for each execution runtime together as a set, we use a random number that is created as one of the very first tasks when conductor is invoked. This 32 bit random number is used in dynamic file naming, directory naming, provider and profile name ID within the files and other things such as salt grains. So we have a consistent random number applied to all runtime execution files and folders that can be used to collect everything and archive the entire execution runtime output when the process if finished.

This unique random number is called the **Cloud Provision ID. Or cpid** when talking about salt grains. Each instance provisioned will get the cpid salt grain (and many other grains) immediately when it's created. This gets into how salt-cloud provides salt specific data to aws for instances that don't exist yet, allowing salt-cloud to bootstrap the new instance with the salt-minion software automagically. This is one of the main benefits of using salt-cloud as the go-between for aws and saltstack. We take advance of built-in features of both technologies without the need to write our own aws library functions.

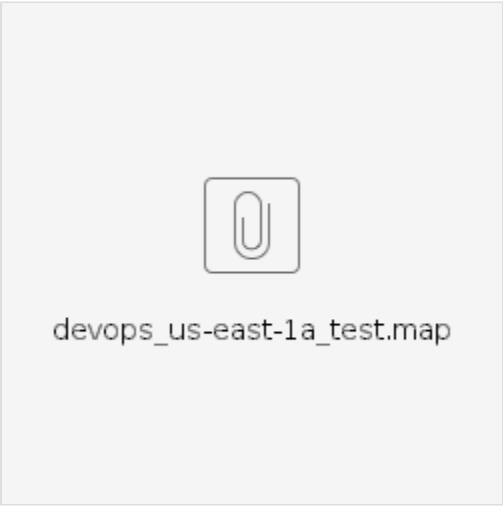
Different configuration key value pairs are set in each of the three files. Salt-cloud uses these and assumes they are all configured properly, the Conductor assures this.

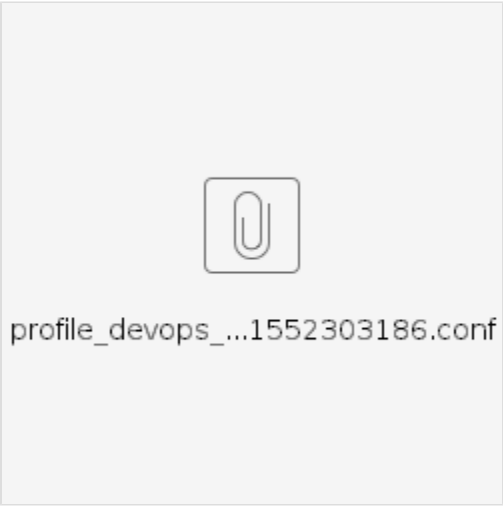
A brief description of the cloud config files: (config file names are also auto-generated using random generator).

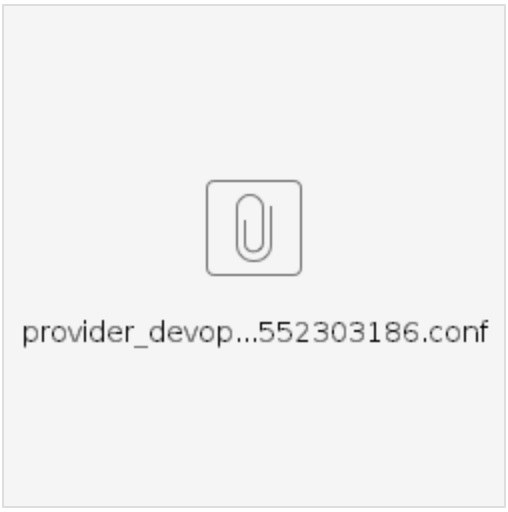
- **provider** - this file contains the highest level of configuration. Items such as aws region and zone. Some configuration can and would be overwritten at a lower level configuration such as the profile or map. However, default values would be set here for things such as whether to create instances with public or private ip, default instance type etc...
    - this file will be created in /etc/salt/cloud.providers.d/
    - the name format will be /etc/salt/cloud.providers.d/*productgroup_CPID.conf*
    - the file is yaml format and contains a top level ID that is referred to by the profile file described below. The provider ID in is also in the form *productgroup_CPID:*
- **profile** - this file contains the next lower level set of configuration. Items such as reference pointer to the provider to use, salt version and script args, minion environment, minions startup state, network interface settings, and some overrides of the provider are possible.
    - this file will be create in /etc/salt/cloud.profiles.d/
    - the name format will be /etc/salt/cloud.profiles.d/*productgroup_CPID.conf*
    - the file is yaml format and contains a top level ID that is referred to by the cloud map configuration file described below. The profile ID is also in the form *productgroup_CPID:*
- **map** - aka, cloud map. this file contains all the instances and every detail for each that will be provisioned such as all needed salt grains, volume configuration, network settings, minion specific settings etc... Some configurations here will override the profile and possibly the provider conf files.
    - this file will be create in /srv/runners/maps/*CPID/*
    - the name format will be /srv/runners/maps*/CPID/productgroup_aws-region-zone_saltenvironment.map*
    - the file is yaml format and contains a tops level ID that is the profile ID described above.

### *Example cloud configuration files*

These files are 100% dynamically generated (including instance names) on each conductor execution. (the yaml format might not be accurate due to wiki upload)






Cloud map Profile conf Provider conf

> The salt conductor command to generate all three files was:
>
> salt-run conduct.group create group=devops system=small_test pillarenv=test region=us-east-1a
>
> *** more on the 'system' option and full details about conductor execution in future documentation*

### *Logging*

There are two types of logging output that is created each time conductor is invoked.

> Conductor has it's own non-session based logging as well. This is a continuous stream of data logged to /srv/runners/logs/conductor/. These are for administrative troubleshooting only.

The log naming and locations are also based on CPID, Cloud Provision ID.

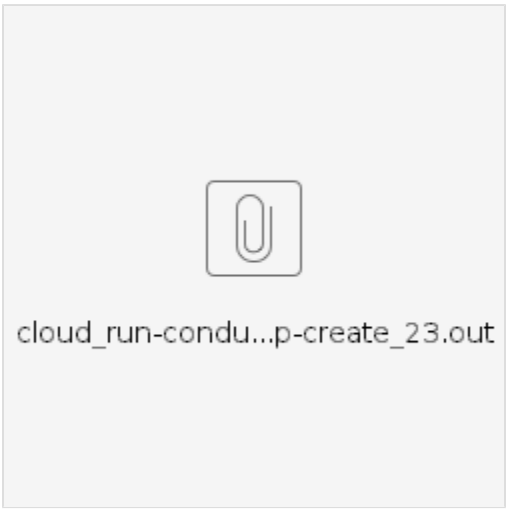
- state_runs
  - for every salt state that is run throughout the runtime of conductor, a unique state output log will be created.
  - all state run output logs for an execution of conductor will be created in the corresponding CPID directory
  - the location is /srv/runners/state_runs
  - example: /srv/runners/state_runs/16833830331552303186

```
drwxr-xr-x. 21 root root 4096 May 24 21:11 .
[[root@saltmaster centos]# ls -altr /srv/runners/state_runs/16833830331552303186
total 640
-rw-r--r--.  1 root root   1831 May 24 20:41 orch_state_run-conductor_common._37742.out
-rw-r--r--.  1 root root   1843 May 24 20:41 orch_state_run-conductor_common._3207.out
-rw-r--r--.  1 root root   1129 May 24 20:49 orch_state_run-conductor_common._49375.out
-rw-r--r--.  1 root root   1842 May 24 20:49 orch_state_run-conductor_common._50330.out
-rw-r--r--.  1 root root 616310 May 24 20:52 state_run-DEVOPS_conductor._137.out
-rw-r--r--.  1 root root   1130 May 24 20:52 orch_state_run-conductor_common._57913.out
-rw-r--r--.  1 root root   1132 May 24 20:52 orch_state_run-conductor_common._20505.out
-rw-r--r--.  1 root root   1843 May 24 20:52 orch_state_run-conductor_common._36321.out
drwxr-xr-x.  2 root root   4096 May 24 20:52 .
drwxr-xr-x. 21 root root   4096 May 24 21:11 ..
```

  - the /srv/runners/state_run/*CPID* directory will contain a log for each state, including orchestration states, that were run. In the above image, you can see 6 orchestration states logs, and one product group log, state_run_DEVOPS_conductor._137.out. In this case the orchestration states were executed from the enabled hooks. See orchestration hooks for more on that.
  - in addition to the unique directory CPID, there is a random number added on to logs within the CPID directory for uniqueness.
  - the state_run_DEVOPS_conductor._137.out log would be the entire salt output from running startup states on all minions that were just provisioned as part of that conductor run. In standard salt json output, the log can be parsed for each new instances output.
  - the orch_state_run_XXXXXXX logs are the output of orchestration states.

> orchestration states execute on the salt master so the log json would look a little different than that of the state logs.

> Non orchestration states are actually executed on the minion, but just initiated from the master. The master receives the output from all minions and coalesces it together. This is native Saltstack behavior and out of scope for this document.

- cloud_runs
  - for each run of the conductor, a single cloud run log is created. The log is actually a text file copy of the return json from aws after provisioning new instances. The conductor receives this return from AWS, objectizes it and uses this to perform additional functionality. In addition to this, conductor creates a copy of this data and dumps it to the cloud_runs/CPID location.
  - the information is highly useful as it provides all the metadata, dynamic key values, and all api details of each instance and it's dependencies
  - the location is /srv/runners/cloud_runs/*CPID*
  - there would only be one file in the unique CPID directory and its format is: /srv/runners/cloud_runs/*CPID*/cloud_run-conductor._group-create_23.out
  - Example file
  - 


> The /srv/runners/state_runs will also contains CPID directories for other conductor action commands such as 'destroy', where nothing is being provisioned, but torn down instead.

All logs can be collected based only on the CPID, and archived somewhere off the salt master if desired.

### *Runtime state of conductor (reservation files)*

Beyond the logging and dynamic configuration file creation described above, one additional requirement for making Conductor parallelization work in multiple executions was to handle the dynamic ID's used for salt grains and the dynamically generated instance names, that have been allocated but do not exist yet.

The high level workflow of conductor is to:

- read Pillar config data to determine what to add to the cloud configuration files it will create
- generate the cloud configuration files
- invoke salt-cloud to create instance and components in aws and bootstrap new minions
- apply salt states

There are other steps, but for describing reservation technique used, this is sufficient information.

Step 2 above, generate cloud configuration files, is why we need to implement a reservation technique in order to achieve error-less parallelized executions of conductor.

The Pillar data that is used contains a few configuration items that help avoid hard coding instance names. Mainly, the nodes, count and basename keys in a provisioning template**.

The basename is a re-usable pattern that is used to create instance names. Teams/groups can create basename patterns as they see fit, but do need to follow some basic standards.

**example of basename in template**

```
devops.activemq:
  ...other config left out for simplicity
  basename: devops-mqXX.REGION.ENV
  nodes: 3
  ...other config left out for simplicity
```

The command would be: ***salt-run conductor.group create role=activemq pillarenv=test region=us-east-1a***

In the example above the Conductor would create cloud configurations and provision 3 instances named as such:

- devops-mq01.us-east-1a.test.foobar.com
- devops-mq02.us-east-1a.test.foobar.com
- devops-mq03.us-east-1a.test.foobar.com <--- foobar.com is an organizational setting in Pillar per environment.

However, if there already was devops-mq01.us-east-1a.test.foobar.com and devops-mq02.us-east-1a.test.foobar.com, the Conductor would generate cloud configuration and provision these:

- devops-mq03.us-east-1a.test.foobar.com
- devops-mq04.us-east-1a.test.foobar.com
- devops-mq05.us-east-1a.test.foobar.com

Essentially the Conductor will do an AWS api lookup in the region AND salt environment that is was told to work in (see command parameter), and create only the lowest (index) instance name that does not exist. So we are guaranteed to get unique names on each runtime execution of Conductor.

** provisioning templates will be discussion in another detailed document so it's out of scope for this document.

---

*You may now see why we need to implement a reservation technique for dynamic names and id's used when Conductor is generated cloud configurations.*

One Conductor runtime execution MUST be aware of all other Conductor runtime executions to the extent that instance names, systems id's, cluster id's and any other identifiers that are dynamically created are not used if another Conductor runtime execution has chosen them for the cloud configuration to be used for instances that haven't been created yet.

Another way to look at it, it's a method to avoid race condition. Conductor checks the AWS environment for 'used' instance names, id's and such, but it also needs to know about names, id's and such that have been selected and created in cloud configuration, but not available yet in AWS.

The implementation...

Beside the unique instance names, Conductor needs to be aware of the following ID's that will become salt grains and thus are used to be very specific when targeting and identifying minions/instances. The reservation is similar to file locking.

The reservations files are created once Conductor has determined that the name or ID doesn't exist in other reservations or AWS environment. The reservations files are removed once the cloud provisioning is complete.

The files are created in the following locations:

- /srv/runners/reserved_names
- /srv/runners/reserved_ids

These are working directories in that when Conductor is doing nothing, there should not be any files in these directories.

The file name formats are as follows: ***(keep in mind some of this is determined by the basename pattern defined in pillar which can be anything the team/group wants within the guidelines)***

***Italic text denotes dynamically created.***

- /srv/runners/reserved_ids
  - *environment_productgroup(team).product(role)*.cluster.id.*INDEX*
  - *environment_cloud.product(role)*.cluster.id.*INDEX*
  - *environment_productgroup(team)*.system.id.*INDEX*
  - *environment*_cloud.system.id.*INDEX*
- /srv/runners/reserved_names
  - *environment_productgroup(team)-role-INDEX*.clid-*ID.region.environment*.domain.com <---- clid-ID automatically added if it's a member of a cluster

> The name format in the about example is somewhat subject to the groups basename pattern in the provisioning template. So this is only an example. The main point is that no matter what any product group/team decides to use as a basename pattern for anything they will be creating, there is no possibility or name, cluster.id, or system.id collisions with anything they or another group are creating.

### Visual look at the runtime state

The following is an images were taken during three simultaneous executions of the Conductor, the second image was taken about 10 minutes later when all three executions were complete. You can see the naming format used by Conductor while it's processing the tasks.

Here are the three separate execution commands issued simultaneously to the Conductor from three separate shell sessions. Each execution is specifying something different to provision, but some things are the same actual roles (machine types) as other executions are requesting. This shows the ability for Conductor to keep track of each execution separately, never mix up things between executions, and dynamically set each instance in each execution to unique names and id's to guarantee accurate targeting. ***More on targeting in the Saltstack technology documentation.***

```
salt-run conduct.group create group=devops role=nifi pillarenv=test
region=us-east-1a
salt-run conduct.group create group=devops role=activemq count=2
pillarenv=test region=us-east-1a
salt-run conduct.group create group=devops system=small_test
pillarenv=test region=us-east-1a
```

> the 'system' in the third command above is actually a logical configuration of a collection of roles to be built all at once. The small_test is actually building a 3 node nifi cluster with 2 activemq instances just to show how separate executions of Conductor can build the same configured instances from the same config without collisions or problems.

During Conductor executions

```
[root@saltmaster centos]# ls -altr /srv/runners/reserved_ids/
total 28
drwxr-xr-x. 8 root root 4096 May 21 19:31 ..
-rw-r--r--. 1 root root    1 May 30 19:21 test_devops.nifi.cluster.id.1
-rw-r--r--. 1 root root    1 May 30 19:21 test_cloud.nifi.cluster.id.1
-rw-r--r--. 1 root root    1 May 30 19:21 test_devops.system.id.1
-rw-r--r--. 1 root root    1 May 30 19:21 test_cloud.system.id.1
-rw-r--r--. 1 root root    1 May 30 19:21 test_devops.nifi.cluster.id.2
-rw-r--r--. 1 root root    1 May 30 19:21 test_cloud.nifi.cluster.id.2
drwxr-xr-x. 2 root root  213 May 30 19:21 .
[root@saltmaster centos]# ls -altr /srv/runners/reserved_names/
total 56
drwxr-xr-x. 8 root root 4096 May 21 19:31 ..
-rw-r--r--. 1 root root   51 May 30 19:21 test_devops-nifimgr-01.clid-1.us-east-la.test.foobar.com
-rw-r--r--. 1 root root   54 May 30 19:21 test_devops-nifi-dummy-01.clid-1.us-east-la.test.foobar.com
-rw-r--r--. 1 root root   48 May 30 19:21 test_devops-nifi-02.clid-1.us-east-la.test.foobar.com
-rw-r--r--. 1 root root   48 May 30 19:21 test_devops-nifi-01.clid-1.us-east-la.test.foobar.com
-rw-r--r--. 1 root root   38 May 30 19:21 test_devops-mq02.us-east-la.test.foobar.com
-rw-r--r--. 1 root root   38 May 30 19:21 test_devops-mq01.us-east-la.test.foobar.com
-rw-r--r--. 1 root root   51 May 30 19:21 test_devops-nifimgr-01.clid-2.us-east-la.test.foobar.com
-rw-r--r--. 1 root root   54 May 30 19:21 test_devops-nifi-dummy-01.clid-2.us-east-la.test.foobar.com
-rw-r--r--. 1 root root   48 May 30 19:21 test_devops-nifi-02.clid-2.us-east-la.test.foobar.com
-rw-r--r--. 1 root root   48 May 30 19:21 test_devops-nifi-01.clid-2.us-east-la.test.foobar.com
-rw-r--r--. 1 root root   38 May 30 19:21 test_devops-mq04.us-east-la.test.foobar.com
-rw-r--r--. 1 root root   38 May 30 19:21 test_devops-mq03.us-east-la.test.foobar.com
drwxr-xr-x. 2 root root 4096 May 30 19:21 .
```

After all three executions completed

```
[[root@saltmaster centos]# ls -altr /srv/runners/reserved_ids/
total 4
drwxr-xr-x. 8 root root 4096 May 21 19:31 ..
drwxr-xr-x. 2 root root    6 May 30 19:29 .
[[root@saltmaster centos]# ls -altr /srv/runners/reserved_names/
total 4
drwxr-xr-x. 8 root root 4096 May 21 19:31 ..
drwxr-xr-x. 2 root root    6 May 30 19:29 .
[root@saltmaster centos]#
```

### *Salt grains operational use case showing the result of Parallel Conductor executions*

As a final look into the result of Conductor building separate but similar role instances from three different user sessions, below is an image that shows the resultant minion/instances salt grains that needed to implement the reservation technique described above.

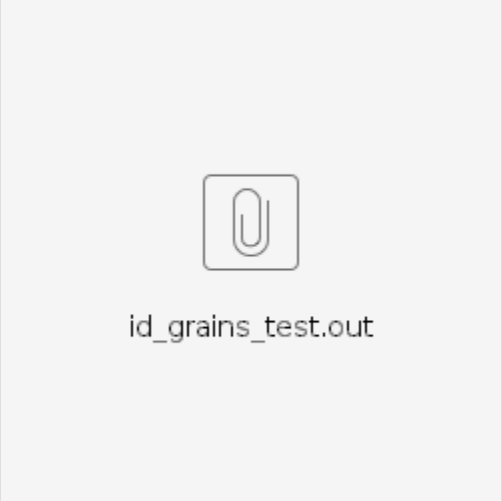
The salt command issues from the master to get this result was:

```
salt '*' grains.item cpid cloud.nifi.cluster.id devops.nifi.clusterid
cloud.system.id devops.system.id
```

Detailed explanation of the Salt command and the salt grains used is out of scope for this document, but to assure the concepts are fully understood, the command breaks down like this in Saltstack terms:

On the salt-master, run a salt command that targets ALL minions (notice salt environment is not in the filter because we can have a single salt-master serve many environments), and retrieve the following three salt grains:

- cloud.nifi.cluster.id (environment wide unique nifi cluster ID)
- devops.nifi.cluster.id (productgroup/team wide unique nifi cluster ID)
- cloud.system.id (environment wide unique system id)  <------ see above note about system. It is one of three types that can be specified in one Conductor execution (out of scope for this document)
- devops.system.id (team wide unique system id)




> Note that in the grains output file above, some minions do not have some of the grains. This was intentional in the command line for each of the three Conductor executions to show how different grains get created based on the provision type that Conductor is being requested. System, Role (single), Role (cluster) are the three types. This is out of scope topic for this document.