Adaptive Management and Administration of IT Infrastructures

Manuel Torrinha

Abstract—With the increased complexity in nowadays Information Technology (IT) environments, as these infrastructures become a mix of physical and virtual, where everything connects to the Internet and security threats propagate almost instantaneously, there is a need to establish models, guides and standards that support this increase to scale. This thesis addresses different aspects of IT Infrastructure management, from Logging events to Monitoring performance to Orchestration and Automation of complex operations, proposing an integration of different tools to work together for managing a complex IT infrastructure in a more "intelligent" and automated way, through a simplified interface, reducing or eliminating the repetition of manual operations tasks, in order to prevent common error cases as well as performance bottlenecks, based, or not, on past occurrences.

Index Terms—cloud orchestration, infrastructure automation, resource monitoring, system management, system administration, information technology

1 Introduction

In an age where Information Technology (IT) infrastructures' dimensions are increasing exponentially (virtually and physically), managing the infrastructure as a whole is crucial. It is vital to have, as close to real-time as possible, information regarding every single component of the infrastructure in such a way that it should be possible to detect changes and diagnose and take actions on the infrastructure programatically.

In a traditional *"reactive"* environment, upon facing a failure, the system either takes actions to remedy that failure or notifies a valid operator, on the nature and status of that failure. For example, when a system (or component) is not responding it is restarted or rolled back to a previous working state.

Another example, is when a certain service can not write data to a specific storage location, due to lack of free space, it could be restarted, resulting in a temporary elimination of the problem (as some service files may get cleared has a result of the restart), but eventually the issue will emerge again.

In a reactive and adaptive environment however, and using the previous example, upon reaching a predetermined threshold on storage free space, an event could then be launched to trigger a *garbage-collector*-like process on the storage Hard Disk Drive (HDD)'s File System (FS), to compress and/delete obsolete or large unneeded files, thus freeing up space.

In either *"reactive"* or *"adaptive"* environments, and picking the above example again, if the system by itself would not be able to restart the service or do a successful compression/cleanup, the system's operator would be notified of the event in order to take human action. However, in the *"preventive adaptive"* environment where the system is made "aware" of what the system's operator does, after that first human intervention, some "learned" steps replicating its actions would be created so that another human intervention would not be necessary anymore in case the same issue would happen again.

• *Manuel Torrinha, nr. 57852,*
E-mail: manuel.torrinha@tecnico.ulisboa.pt,
Instituto Superior Técnico, Universidade de Lisboa.

October, 2017.

1.1 Objectives

This work's objective is to present and implement a *"preventive adaptive"* environment to empower the management and administration of IT Infrastructures, in which a set of tools are integrated to achieve the following goals:

- Manage programmatically any component of an IT Infrastructure (networks, networked devices, compute systems, storage systems and networks);
- Monitor any component of an IT Infrastructure;
- Manage the associated information and meta-data;
- Do all the above remotely from any point on the Internet, in a secure fashion.

1.2 Environment

This work's environment is composed of three distinct layers:

- **Operation**
- **Control**
- **Infrastructure**

In the **Operation** layer one can consider the Operator(s)/Administrator(s) and/or Tenants that can manage parts (or the sum of them) of the infrastructure, be it via a web/graphical interface or via an Application Program Interface (API).

The **Control** layer is where the Main Controller, responsible for all of the infrastructuresĆontrollers, is located. One may interact directly from within the controller instead of using the previously mentioned interfaces.

The **Infrastructure** layer is where the controller for each infrastructure is located, and which send instructions to the controlled components (from now on referred to as minions).

2 Related Work

Recent advances in IT infrastructures, namely for "Cloud-based" Services, where scalability, elasticity and availability are key factors, have also seen advances in the way Management and Administration of these critical infrastructures are achieved, be it on log analysis and systems monitoring or on configuration management and automation. This is a rising subject, with exponential development evolution as new tools with different approaches have emerged in the last few years. This section presents and discusses some of the relevant publications covering these topics in different areas, from cloud environments [1]–[3], scientific grid computing environments [4] to smart home environments [5].

Delaet *et al.* [6] provide a framework for evaluating different System Configuration tools. They also define a conceptual architecture for these tools, illustrated in Figure 1. In essence, this type of tool is composed by a "master" element (typically residing in a deployment node) containing a set of specialized modules that "translate" through some form of manifest files, the specific configuration for each component of the remote managed nodes of the infrastructure. Each remote managed node, through some form of local "agent" conveys to the "master" detailed information about the node and executes configuration actions determined by the "master". The "master" compiles in a repository a catalog (inventory) with information on the nodes and on how they should be configured.

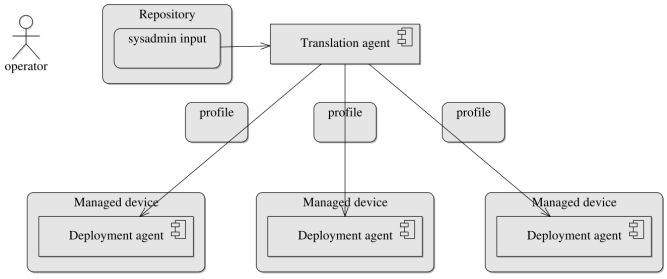


Figure 1. A conceptual architecture of system configuration tools [6]

In their paper, John Benson *et al.* [1], address the challenge of building an effective multi-cloud application deployment controller as a customer add-on outside of a cloud utility service using automation tools such as Ansible (ansible.com), SaltStack (saltstack.com) and Chef (www.chef.io) and compare them to each other as well as with the case where there is no automation framework just plain shell

code. The authors compare those three tools in terms of performance in executing three sets of tasks (installing a chemistry software package, installing an analytics software package, installing both software packages), the amount of code needed to execute those tasks and different features they have or do not have, such as a Graphical User Interface (GUI) and the need for an agent.

Ebert, C. *et al.* [7] address and discuss this recent organization culture coined **DevOps**, namely the toolset used at different phases, from Build, to Deployment, to Operations. The authors conclude that "DevOps is impacting the entire software and IT industry. Building on lean and agile practices, DevOps means end-to-end automation in software development and delivery." ([7]), with which I agree on. The authors also state that "Hardly anybody will be able to approach it with a cookbook-style approach, but most developers will benefit from better connecting development and operations." ([7]) with which I do not totally agree with, as the work here proposed will try to demonstrate, by showing that a reactive infrastructure can be created to act autonomously on certain events, without leaving the cookbook-like approach, but through a set of triggered events that can be used to automate software development and delivery. There are already tools, such as **SaltStack** that makes possible for certain events to intelligently trigger actions, allowing therefore IT operators to create "autonomic" systems and data centers instead of just static runbooks. The authors also state that a "mutual understanding from requirements onward to maintenance, service, and product evolution will improve cycle time by 10 to 30 percent and reduce costs up 20 percent." ([7]) pointing out that the major drivers are fewer requirements changes, focused testing, quality assurance, and a much faster delivery cycle with feature-driven teams. They also point out the large dynamics of this **DevOps** culture as "each company needs its own approach to achieve DevOps, from architecture to tools to culture." ([7])

In a recent whitepaper release by **SaltStack** they state that "IT teams are already stretched to the limit as they try to keep data center resources secure and running efficiently. Attempting to meet the challenge without automation is virtually impossible." ([8]) – which clearly shows the need for this type of solution, as these environments are constantly changing, so much that they require IT professionals to make use scalable and intelligent configuration automation and orchestration to the applications and resources, often running on multiple public clouds and in-house infrastructures.

Saltstack took a different approach for its architecture, illustrated in Figure 2, when compared to the architecture defined by Delaet *et al.* [6], illustrated in Figure 1. Comparing both, there are some similarities but it can also be seen that there is a bidirectional channel of communication between the managed devices and the Salt Master which corresponds to an Event Bus that connects to different parts of the tool. The Runner is used for executing modules specific to the Salt Master but can also provide information about the managed devices. The Reactor is responsible for receiving events and corresponding them to the appropriate state or even to fire another event as a consequence of the former. As stated in their documentation, "Salt Engines are long-running, external system processes that leverage Salt"[1], allowing to integrate external tools into the Salt system, for example to send to the salt Event Bus messages of a channel on the **Slack** (slack.com) communication tool [2].

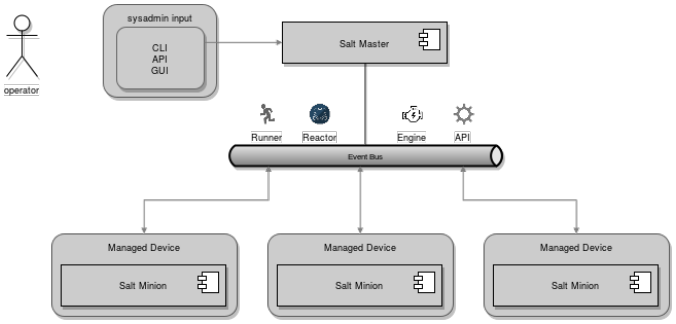


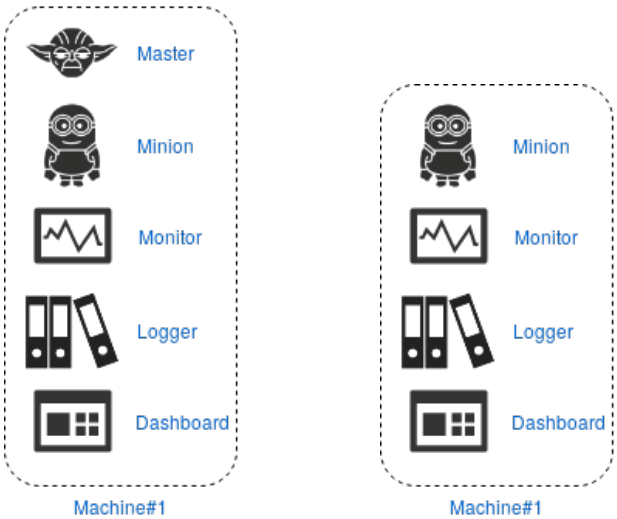
Figure 2. Saltstack architecture

1. https://docs.saltstack.com
2. https://github.com/saltstack/salt/blob/develop/salt/engines/slack.py

3 ARCHITECTURE

For this work we will be using Saltstack's orchestration tool *Salt* as from my personal and professional experience it is best adequate to aid in the implementation of this work's objectives, as its requirements for a device to be managed by it can be as little as having a REpresentational State Transfer (REST) API through which one can interact with. Its overall architecture can be seen in Figure 2. In this section we will see this thesis' architectural, network and hardware requirements, as well as *Salt*'s software requirements and how to use *Salt* as a full infrastructure management tool.

The minimum configuration for creating this work's infrastructure with Salt is one machine, which takes the role of Master, Minion, Logger, Monitor, and Web Interface (Dashboard), see Figure 3. Although this does not achieve much, it still allows to test operations, be it configuration management, reactions to certain events, monitoring of processes or to just do some basic testing of this tool.



(a) *with* Master (b) *without* Master

Figure 3. Monolithic example

Consider the two scenarios represented in Figure 4:

1) an *Operator* within the *Control* layer
2) an *Operator* outside the *Control* layer, with *Control* and *Infrastructure* layers behind a Network Address Translation (NAT)

In the first scenario, the Operator is either within the Main Controller itself or in the network neighbourhood, meaning that it has network access within the private network of which the Main Controller is part of.

In the second scenario, the Operator and the Main Controller are in distinct networks, meaning that the Operator with have to have external access to the infrastructure, either using the Main Controller has a bastion server, or by assigning the minions with a public Internet Protocol (IP) address. We will see that as public IP addresses are not that abundant, using a bastion becomes a necessity.

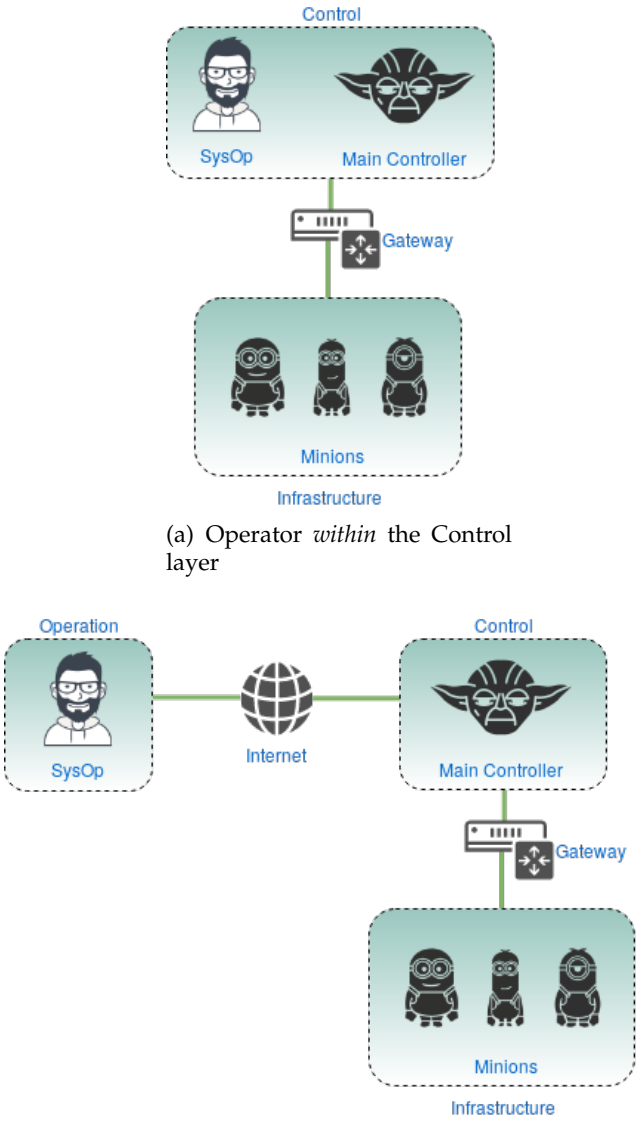
The Salt administration tool, by Saltstack, has several key components which make it a complete framework for managing and administering an IT infrastructure. In each of the following subsections we will see what they are used for and how to make use of them.

"Running pre-defined or arbitrary commands on remote hosts, also known as remote execution, is the core function of Salt" [9]. Remote execution in Salt is achieved through *execution modules* and *returners*. Salt also contains a configuration management framework, which complements the remote execution functions by allowing more complex and interdependent operations. The framework functions are executed on the minion's side, allowing for scalable, simultaneous configuration of a great number of minion targets.

"The Salt Event System is used to fire off events enabling third party applications or external processes to react to behavior within Salt" [10]. These events can be launched from within the Salt infrastructure or from applications residing outside of it.

To monitor non-Salt processes one can use the *beacon system* which "allows the minion to hook into a variety of system processes and continually monitor these processes. When monitored activity occurs in a system process, an event is sent on the Salt event bus that can be used to trigger a reactor" [11].

The pinnacle of the two previous components is using that information to trigger actions in response to those kinds of events. In Salt we have the *reactor system* "a simple interface to watching Salt's event bus for event tags that match a given pattern and then running one or more commands in response." [12].



(a) Operator *within* the Control layer

(b) Operator *outside* the Control layer

Figure 4. Network Architecture

Most of the information may be static, but it can belong to each individual minion. This information can be either gathered from the minion itself or attributed to it by the Salt system. The first "is called the *grains* interface, because it presents salt with grains of information. Grains are collected for the operating system, domain name, IP address, kernel, OS type, memory, and many other system properties." [13]. The latter is called the *pillar*, which "is an interface for Salt designed to offer global values that can be distributed to minions" [14], in a private way relative to every other minion, if need be.

3.1 The Top File

An infrastructure can be seen as being an applicational stack by itself, by having groups of different machines set up in small clusters, each cluster performing a sequence of tasks. The Top file is were the attribution of configuration role(s) and Minion(s) is made. This kind of file is used both for State and Pillar systems.

Top files can be structurally seen has having three levels:

Environment (ϵ) A directory structure containing a set of SaLt State (SLS) files

Target (Θ) A predicate used to target Minions

State files (Σ) A set of SLS files to apply to a target. Each file describes one or more states to be executed on matched Minions

These three levels relate in such a way that $\Theta \in \epsilon$, and $\Sigma \in \Theta$. Reordering these two relations we have that:

- Environments contain targets
- Targets contain states

Putting these concepts together, we can describe a scenario in which the state defined in the SLS file 'ntp.sls' is enforced to all minions.

Listing 1. Example Top File

```
base:
  '*':
    - ntp
```

3.2 Environments

When executing a state, the default state environment considered is *base*, we are able to define extra environments and/or add directories to already existing environments. In the Master config file we set a key, *file_roots*, which defines the root location(s) to be used by the file server. The file server can aggregate SLS files from different locations, in this case the states can not be matched among the different directories as to ensure the reliability of delivering the correct state file.

3.3 Event System

In Figure 2 we can see the Reactor element, this element is part of a more broad system, the Event System (represented in Figure 5) which is 'used to fire off events enabling third party

applications or external processes to react to behavior within Salt' [10]. The Event System has two main components:

Event Socket ⇒ from where events are published

Event Library ⇒ used for listening to events and forward them to the salt system

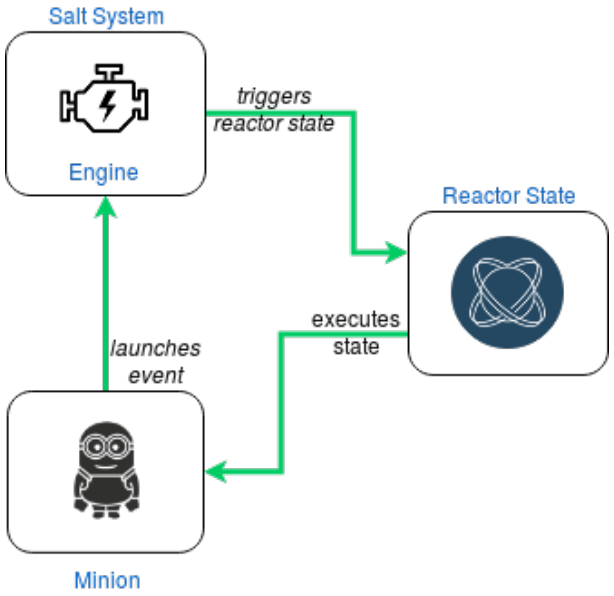


Figure 5. Saltstack Event System Flow Example

'The event system is a local ZeroMQ PUB interface which fires salt events. This event bus is an open system used for sending information notifying Salt and other systems about operations' [12].

This associates SLS files to event tags on the master, which the SLS files in turn represent reactions. This means that the reactor system has two steps to work properly. First, the reactor key needs to be configured in the master configuration file, this associates event tags SLS reaction files. Second, these reaction files use a Yet Another Markup Language (YAML) data structure similar to the state system to define which reactions are to be executed.

Besides the tag, each Salt event has a data structure, a dictionary which can contain arbitrary information about the event.

Similar to other components of Salt, Reactor state are interpreted in YAML by default, although they can be written in other formats, we will keep using YAML for markup with Jinja templating when needed.

The structure for Reactor SLS files resembles State SLS files in that each block of data starts with an unique Identifier (ID), followed by a function and any arguments to be passed to that function.

A simple reactor state can be seen in Listing 2. This reactor is used when the Master receives an event with the *salt/minion/<minion_id>/start* tag, and it executes the *highstate* function.

Listing 2. highstate.sls Reactor State

```
highstate_run:
  cmd.state.highstate:
    - tgt: {{ data['id'] }}
```

4 Development

A *salt-cloud* tool in the SysOp Workstation was used to launch the Master (bastion) within the Openstack Cloud Provider. Before we can do this we need to specify which providers exists in our environment, by setting up the file */etc/salt/cloud.providers.d/<provider>.conf*, where '<provider>' is the provider name. Then we must describe in the */etc/salt/cloud.profiles.d/<profile>.conf* file the machine types that can exist in our infrastructure. Finally, we define in a map file all the machines that are to be created when invoking the command `salt-cloud -m <infrastructure>.map`

The *salt-ssh* tool was used to issue commands and apply Salt States to the infrastructure without maintaining a permanent connection to either the bastion host or the rest of the infrastructure (with public IPv4 addresses limited to 1). The minimal configuration for salt-ssh to work is:

Saltfile ⇒ defines which configuration files and directories are to be used, extra option can be passed to the *salt-ssh* tool.

roster ⇒ defines the minion infrastructure, as in this case the salt-ssh minion is unaware of its "master".

master ⇒ just like the master config file for *salt-master*, mainly used to define from where the SLS files are read.

states ⇒ where the Top File and the SLS files are located.

After setting up the aforementioned Workstation (see **??**) we can start to write the manifests that will be executed, also known as SLS files.

We can define the development process as a step by step process, as follows.

First we describe what actions we want to happen on a minion, let us take as an example the flow in Listing 3 in a top-down approach:

Objective ⇒ to ensure a functional running Network Time Protocol (NTP) client

Requirements

- the NTP client software must be installed
- the NTP client configuration file must be in a predetermined state
- the NTP client service should be stopped when setting up the configuration file
- the NTP client service is to be running after the state completes

Listing 3. NTP SLS file

```
{% set ntp_conf = pillar['ntp']['ntp_conf
    ↪ '] %}
ntp:
  pkg.installed
  service.running:
    - enable: True
    - watch:
      - file: {{ ntp_conf }}
{{ ntp_conf }}:
  file.managed:
    - template: jinja
    - source: salt://env/ntp/etc/ntp.conf
      ↪ .tmpl
    - mode: 644
    - user: root
    - require:
      - pkg: ntp
ntp-shutdown:
  service.dead:
    - name: ntp
    - onchanges:
      - file: {{ ntp_conf }}
```

After defining these requirements we can start writing the SLS file by using the proper State Modules and state keywords for state ordering. In this example we use the *pkg.installed* state module function which maps to the system's package manager, the *service.running* for making sure that the service is left running (the *watch* keyword makes sure that the state in run only if there were changes to the watched state). We then make use of the *file.managed* state module function to generate the configuration file based on a template file. One thing to note is the *service.dead* that uses the *onchanges* keyword, this makes sure it runs **before** the state it is "watching" is executed.

Now we can test the manifest with the command `salt "*" state.apply ntp test=True` and if everything goes as expected we can apply it definitely without the *test* keyword, `salt "*" state.apply ntp`.

5 Evaluation

In Figure 6 we can see the actual setup used in this thesis' evaluation. The model used was the one referred to in Figure 4(b). An OpenStack Cloud Controller was used to deploy the whole infrastructure in to, in which the Operator is outside the Control layer network.

The following are the steps taken to get the whole infrastructure online:

1) Using *salt-cloud* using the map file in Listing 4, we launch the bastion instance (master1 in Figure 6) in the OpenStack provider. This also makes sure the bastion is provisioned with salt-master with the 'make_master' keyword.
2) At this time the master is online but has no SLS files, using *salt-ssh* we configure the SLS files in the bastion, by applying the 'bootstrap.states' SLS.
3) After the previous state runs successfully, we can restart the *salt-minion* service on the bastion. As soon as the *salt-minion* is launched and ready to receive commands from the master, it will send a 'minion_start' event, which will trigger the 'highstate.sls' reactor state and run the states matching the minion_id configured in the Top File.
4) Now that the bastion/*salt-master* is set up, we can start our infrastructure by using *salt-cloud* with the map file shown in Listing 5, from within a Secure Shell (SSH) session or using the 'cmd.run' execution module function. The minion instances should

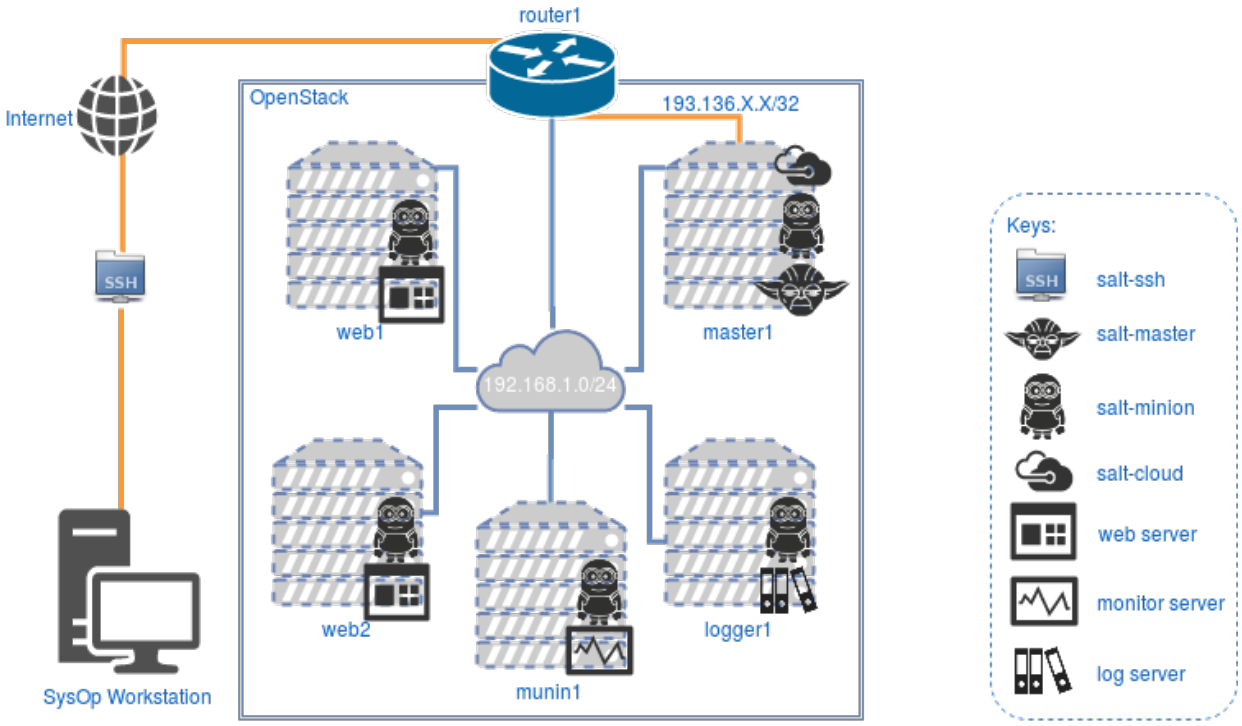


Figure 6. Test Environment

be launched and provisioned via the *salt-cloud* command.

5) Similar to step 3, the minions start up and and send a 'minion_start' event, triggering the 'highstate.sls' reactor state, leaving the minions in the state configured by the Top File, and ready to be sent commands to via the *salt* command (from within the bastion).

Listing 4. bastion map file

```
master_512:
  - master1:
      make_master: True
      ssh_interface: public_ips
      ssh_username: ubuntu
```

Listing 5. instances map file

```
minion_web_512:
  - web1
minion_munin_512:
  - munin1
minion_log_256:
  - logger1
minion_256:
  - nfs1
```

Three sets of tests were designed in order to test different parts of this system.

The first relates to Virtual Machine (VM) creation within the cloud provider, it evaluates how long it takes for the OpenStack framework to make sure a predetermined number of VMs, defined in map files do not exist and afterwards launching them again from scratch.

The second set tests for execution times of Salt Execution Modules and Salt State Modules, since the command is issued on the master and until all targeted minions return, which are:

- execution module *test.ping*, which makes the minion send a simple return 'True' to the master.
- SLS env.hosts, which configures the '/etc/hosts' file, on all minions.
- SLS env.salt.cloud on the bastion minion 'master1', used to set up the bastion host as well.
- SLS env.salt.minion, which installs the *salt-minion* package and ensures the configuration is as programmed.
- every SLS that each target minion has been mapped to in the top file.

The third set tests the response time between the event being fired and the reactor state response to reach the minion, for this we used auxiliary scripts to interact with the Salt

API. One to listen to any event arriving in the Master message bus, and another to listen to a specific event in the Master message bus.

Besides the first set, which was run 10 (ten) times, every test was run 100 (one hundred) times, and we presented the average, median, minimum and maximum Time To Complete (TTC) for each in Table 1. The amount of iterations for each test were these due to system and time constraints, the more tests we could have run the more data we would have to compare results.

The data files were read using an auxiliary Python script, from which the aforementioned statistical data was extracted.

In Table 1 we can see that the results deviate a lot, which makes the results have less significance, there are differences in tests which go to the order of 4000% (env.hosts). These were tests with operations like simple file modifications, typically one line in a less than ten lines file, which do not justify these time differences.

Equation (1) represents the standard deviation for the TTC values of each test:

σ Represents the standard deviation from the mean TTC.
N Is the total number of test runs.
x_i is the actual TTC a run i took.
$\bar{x}$ is the mean TTC.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N-1}} \quad (1)$$

6 Conclusion

We saw how we can launch several different machines at the same time, and how to get them bootstrapped to our Saltstack system.

We analysed the different Saltstack components, namely:

- Execution Modules
- State Modules
- SLS files
- Event System
- Pillar System

We saw how to use *salt-cloud* to create and destroy VM instances within the Openstack framework, how to use *salt-ssh* to manage minions that do not have a salt agent running and a brief view on how to manage devices which have close to none computacional capabilities ('dumb' devices). Each implemented SLS state was presented, as well as how to use it in different ways, be it by means of a direct call, defining it in the Top File and running the *highstate* function, or by means of a trigger to an event.

The tests performed where related to time taken to execute an action or a set of actions, the successfulness of the tests was satisfactory in the way that all of the tests finished successfuly. However, the time taken for each test to execute was not satisfactory, as the results were sometimes far apart, as we can see by the σ TTC values in Table 1.

However, we can conclude that *salt* is adequate for managing programmatically any component of an IT Infrastructure (networks, networked devices, compute systems, storage systems and networks), monitor said components and also manage information and metadata associated with these devices, in a secure fashion, regardless of having the requirements to run a management agent. Moreover the results regarding the reactor benchmark (Table 1) were fast enough for its execution times to be discarded, showing added value of making use of reactor states besides executing states manually. It is to note the discrepancy of values between runs, which shows that the provided OpenStack framework may have not been the most adequate for this system, taking into account the minimum requirements refered, the VMs made available to deploy had at maximum 1 Central Processing Unit (CPU) and 512MB of Random Access Memory (RAM). As a workaround for the ram requirements, a swapfile within each of the VM's filesystem was created and mounted, using the SLS env.swap, see. No workaround was possible regarding the CPU limitations. This limited the amount of machine that could be launched, as well as the possibility of demonstrating more functionalities in a feasible schedule. Other VMs being executed within the provided Openstack system could have also hindered the tests performance.

Although no dashboard was implemented, some light was shed on how one would be

Table 1
Test Results

Benchmark: cloud destroy→create	#Minions	#Runs	Average TTC seconds	Median TTC seconds	Minimum TTC seconds	Maximum TTC seconds	σ TTC seconds
T1	4	10	1394.559	1282.457	1233.402	2278.470	316.965
T2	3	10	1270.986	1012.512	884.689	2695.600	638.226
T3	2	10	984.4976	688.027	620.071	2884.033	687.806
T4	1	10	510.621	471.646	347.945	764.963	152.451
Benchmark: states state @ minion	**#Minions**	**#Runs**	**Average TTC seconds**	**Median TTC seconds**	**Minimum TTC seconds**	**Maximum TTC seconds**	**σ TTC seconds**
test.ping @ "*"	5	100	1.007	0.932	0.867	2.270	0.234
env.hosts @ "*"	5	100	8.476	4.808	1.578	60.333	10.105
env.salt.cloud @ "master1"	1	100	15.594	12.380	2.994	49.819	9.446
env.salt.minion @ "*"	5	100	44.319	39.731	27.892	111.373	16.638
highstate @ "*"	5	100	108.018	101.119	44.602	200.184	34.326
Benchmark: reactor event→state	**#Minions**	**#Runs**	**Average TTC seconds**	**Median TTC seconds**	**Minimum TTC seconds**	**Maximum TTC seconds**	**σ TTC seconds**
minion_start → highstate	1	100	—	—	—	—	—
salt/auth → auth_pending	1	100	—	—	—	—	—
inotify → env.salt.minion	1	100	—	—	—	—	—

designed, which functions could be performed on the managed devices and what implies implementing one.

In spite of trying to setup Logger and Monitor minions, none were implemented as the overall infrastructure performance would be severely impacted as we can see from the tests already performed, in Table 1, where no such minions exist and the system is already slow.

REFERENCES

[1] J. O. Benson, J. J. Prevost, and P. Rad, "Survey of automated software deployment for computational and engineering research," *10th Annual International Systems Conference, SysCon 2016 - Proceedings*, 2016.

[2] Y. Demchenko, S. Filiposka, M. d. Vos, and ..., "ZeroTouch Provisioning (ZTP) Model and Infrastructure Components for Multi-provider Cloud Services Provisioning," 2016.

[3] N. K. Singh, S. Thakur, H. Chaurasiya, and H. Nagdev, "Automated provisioning of application in IAAS cloud using Ansible configuration management," *Proceedings on 2015 1st International Conference on Next Generation Computing Technologies, NGCT 2015*, no. September, pp. 81–85, 2016.

[4] E. Imamagic and D. Dobrenic, "Grid Infrastructure Monitoring System based on Nagios," *Proceedings of the 2007 Workshop on Grid Monitoring*, pp. 23–28, 2007.

[5] L. Yao and U. Australia), "Context as a Service: Realizing Internet of Things-Aware Processes for the Independent Living of the Elderly," in *Service-Oriented Computing*, 2016, no. October, pp. 763–779.

[6] T. Delaet, W. Joosen, and B. Vanbrabant, "A survey of system configuration tools," *International Conference on Large Installation System Adminstration*, pp. 1–8, 2010.

[7] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "DevOps," vol. 33, no. 3, pp. 94–100, 2016.

[8] Saltstack, "SaltStack Enterprise 5.0 (White Paper)," 2016. [Online]. Available: https://saltstack.com/saltstack-enterprise-white-paper/

[9] ——, "Remote Execution - Saltstack Documentation - v2017.7.1," 2017. [Online]. Available: https://docs.saltstack.com/en/latest/topics/execution/index.html

[10] ——, "Event System - Saltstack Documentation - v2017.7.1," 2017. [Online]. Available: https://docs.saltstack.com/en/latest/topics/event/events.html

[11] ——, "Beacons - Saltstack Documentation - v2017.7.1," 2017. [Online]. Available: https://docs.saltstack.com/en/latest/topics/beacons/index.html

[12] ——, "Reactor System - Saltstack Documentation - v2017.7.1," 2017. [Online]. Available: https://docs.saltstack.com/en/latest/topics/reactor/index.html

[13] ——, "Grains - Saltstack Documentation - v2017.7.1," 2017. [Online]. Available: https://docs.saltstack.com/en/latest/topics/grains/index.html

[14] ——, "Pillar - Saltstack Documentation - v2017.7.1," 2017. [Online]. Available: https://docs.saltstack.com/en/latest/topics/pillar/index.html