A Comparison of Configuration Management Tools in Respect of Performance and Complexity

Dave Hill

***Abstract*—Cloud computing has brought many benefits to business, one of the key being the ability to build resilience into their IT infrastructure. This resilience comes with a cost though, as the time utilised by these resources is what the cloud providers charge for. As a result, optimising the time any resources are alive is a key component of an architecture strategy. This research project will aim to test several methods of bootstrapping resources for use and document the findings in order to make educated decisions regarding software deployments and overall architecture design. The tools under test are Ansible, Chef, Puppet and Salt and with this work, a test framework is also made available to allow for future work to build off these findings.**

***Index Terms*—Infrastructure, Infrastructure as code, IaC, Terraform, AMI, AWS, Infastructure Automation, Configuration Management, Chef, Puppet, Ansible, Salt, Terraform, Azure**

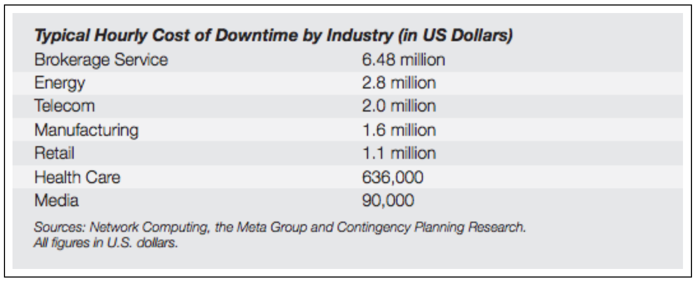
I. Introduction

INFRASTRUCTURE Automation is one of the most prevalent consequences of the inevitable shift of the IT industry to cloud computing. With computing hardware now available on demand with relatively simple interfaces to provision environments, this is an area that has become synonymous with modern computing architecture and it's impact has been evident throughout a huge number of studies and reviews. As capabilities and platforms have grown, mechanisms to control cloud-based hardware have evolved and raised the standard of rapid deployments and updates.

Configuration Management is understood as automating the implementation, and maintenance, of a desired state for provisioned hardware. Although this predates the move towards cloud computing, the shift towards Infrastructure as a Service (IaaS) has reinforced the importance of ensuring that on-demand resources are not just initialised, but configured to meet the needs of the software deployed to them. Typical use-cases range from security and user management, to dependency installation and even validation of hardware provisioning.

A. Motivation

Companies with an online presence will agree that downtime needs to be avoided at all costs. This is why such bold claims from services like Amazon are seen, that state their S3 services will be available 99.99% of the time (Amazon, 2019). A Gartner survey from 2014 stated the average cost per minute of downtime was \$5600 (Lerner, 2014). In Figure 1 the estimated costs per industry can be seen. While is unlikely to see differences of hours throughout this study, these figures are indicative of the potential that can be realised by further optimising these processes. For example, in the case of the Brokerage Service in Figure 1, a second of downtime can cost \$1800 so saving 4 or 5 seconds for every deployment can make a huge financial impact. This is likely to continue to get more expensive as e-commerce and digital services continue to grow as seen in recent years (Oladapo and Onyeaso, 2018).



Typical Hourly Cost of Downtime by Industry (in US Dollars)	
Brokerage Service	6.48 million
Energy	2.8 million
Telecom	2.0 million
Manufacturing	1.6 million
Retail	1.1 million
Health Care	636,000
Media	90,000

Sources: Network Computing, the Meta Group and Contingency Planning Research. All figures in U.S. dollars.

Figure 1. Downtime Costs per industry (Plant, 2014)

B. Contribution

The intended goal for this research is to highlight the differences in performance across the various tools used in a series of tests. This, coupled with the secondary set of metrics of cost and usability, is intended to provide an insight into the impact of each tool. Not all tools, or platforms are tested due to limitations in time and a shrinking return on investment when looking at more obscure technologies. This research can document the results of the exact tests run, but any insights beyond actual tests are merely inferred from these results.

II. Literature Review

Configuration management has unquestionably become significantly more popular since the surge towards cloud computing and Infrastructure-as-Code. its roots can be traced back to the early 90s where CFEngine (http://cfengine.com) was first conceived by Mark Burgess who used it to maintain Unix machines he was responsible for (Cowie, 2014). While this was certainly a front-runner to many of the successive tools that followed, long before the leap to cloud computing, there were several tools available in what Basher (2019) referred to as the ”iron age” of computing, referring to the physical racks holding machines that still needed to be maintained.

With their prototyping tool, DRAT (Deployment Recovery and Automation Technology), Klein and Reynolds (2019) using a series of python scripts, developed the capability to automatically generate configuration management scripts for an specific existing infrastructure. The process flow used by it can be seen in Figure 2. It begins it's construction by first scraping the data from each machine's package manager to understand what libraries were installed on it. It also

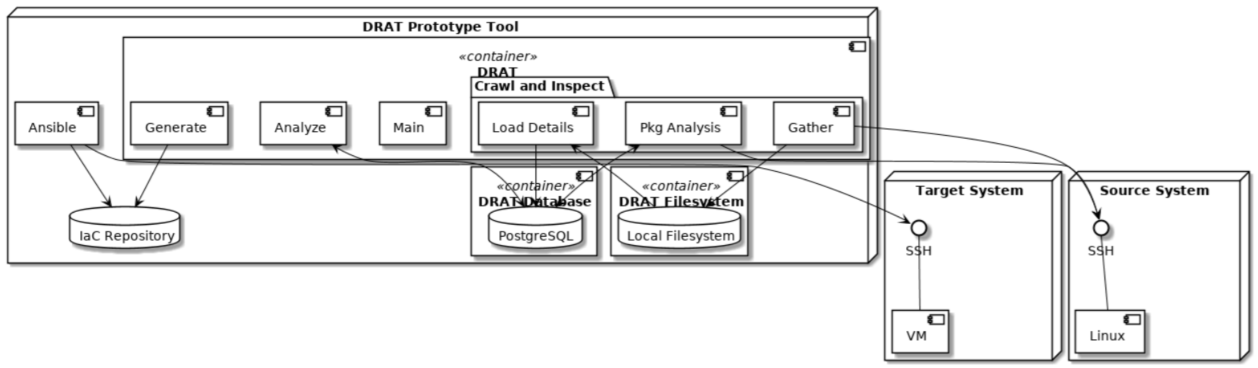


Figure 2. Process Flow for DRAT tool (Klein and Reynolds, 2019)

searched through a series of files paths to grab associated configuration files corresponding to each of these libraries. Once this analysis was done, the tool would then automatically generate Ansible artifacts with the intention establishing a foundation for systems not currently using any configuration management options. The concept itself is not one that is inherently complex, instead being more time consuming to maintain.

An interesting survey of system configuration management tools was identified which took 11 configuration management tools with the intentions of comparing them across multiple criteria (Thomas, Joosen, and Vanbrabant, 2010).

Tool	Version
BCFG2	1.0.1
Cfengine 3	3.0.4
Opscode Chef	0.8.8
Puppet	0.25
LCFG	20100503
BMC Bladelogic Server Automation Suite	8
CA Network and Systems Management (NSM)	R11.x
IBM Tivoli System Automation for Multiplatforms	4.3.1
Microsoft Server Center Configuration Manager (SCCM)	2007 R2
HP Server Automation System	2010/08/12
Netomata Config Generator	0.9.1

Figure 3. Tools compared by (ibid.)

As seen in Figure 3, many of the tools used were proprietary and commercial configuration management solutions from leading companies in the IT industry. As of today, a number of these tools have ceased formal releases, opting to move their functionality to open source platforms such as github to be maintained by heavily invested or passionate individual contributors. Only Chef and Puppet are present in this study which makes up only half of the tools intended to be used in this research project. The study targeted elements of each tool such as deployment models, syntactical input and UIs, configurable and conditional abilities, documentation, scalability, workflows, distribution mechanisms, usability, monitoring, versioning, security, commercial support and maturity. Disregarding usability, the remainder of this list of comparison points are almost universally available through documentation provided by each tool, indicating that this survey was done without attempting to utilise any of these tools, which the team do not explicitly state that was done or required.

A similar study (Önnberg, 2012) but based just on Chef, Puppet and CFEngine was a little more investigative in the technical aspects of each of these tools. In his research Önnberg (ibid.) went through installation steps and dependencies of each of the tools, as well as some of the attributes covered by Thomas, Joosen, and Vanbrabant (2010). The focus was narrowed to focus with greater clarity on the technical details of these tools with observations around the release package format and alignment with the installation documentation but this was the depth of the technical review and this once again indicates that performance based assessments of these tools is not easily found in academic literature.

III. Research Design

The tools chosen for this research are Chef, Puppet, Ansible and Salt. These were chosen as they were identified as a selection of the most popular tools currently available with a large online presence (Johari, 2019) to allow for further research, support and information around compatibility. In order to provide a balanced insight into the impact the tools will have, the tests will use multiple cloud computing platforms. In this case, Microsoft's Azure and Amazon Web Services (AWS) were selected to provide a mechanism to provision computing instances on demand and at scale as needed. By using both platforms, this should rule out any provider specific implementations that may benefit one tool over another. Any significant deviations between the performance of the same tools, executing the same tests, with only the cloud provider differing, could indicate back end configuration more favourably implemented for a specific tool set or underlying technology.

A. Planned Approach

The intended approach for this research is to capture specifically the performance of each tool in establishing how to get a standard machine in it's desired state. The process of provisioning infrastructure, the speed of which is *not* a metric that will be tracked or reporting on, is still critically important in order to perform the experiments. Once the machines are provisioned, they will need to connect to their relevant master node and for each test, there will be a different set of configuration files needed to complete the experiment.

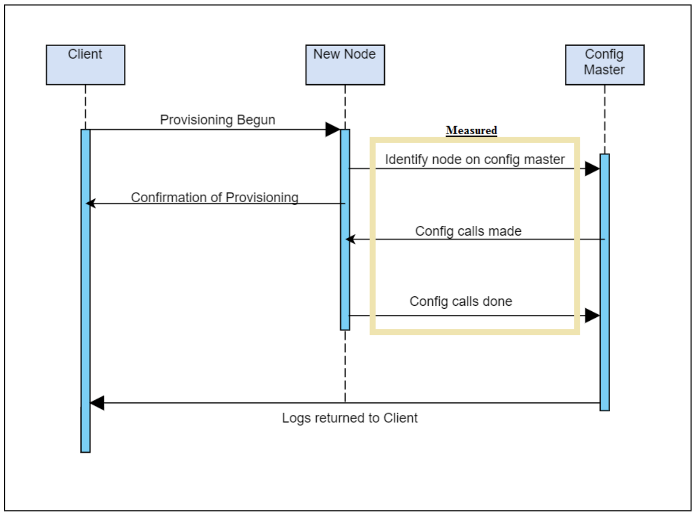


Figure 4. Provisioning steps targeted as part of this study

Once the configuration has completed on the node, a log will capture the time needed to execute the test and these logs will then be collected for analysis. As seen in Figure 4, only tracking the time taken to configure the node into the desired state is captured, provisioning the node from the cloud provider is not considered part of the timings. Once the node is provisioned and available from the cloud provider, the timings will begin. Once the configuration is successfully applied *and* reported back to the master, the timing will stop. As an aside to this research, there will also be an investigation into the usability and ease of installation for each of the tools. If a tool is marginally faster to execute but the learning curve is much higher and integration much more difficult, this additional complexity is worth noting before adopting any findings based off of this study.

IV. Implementation

A. Provisioning

Terraform scripts were written to create the configuration management master nodes and this required some additional bootstrapping to ensure they were created ready to start operating as a master with minimal manual intervention. For the purposes of these tests, the "`user data`" provisioner of AWS and "`remote-exec`" provisioner of Azure were used to execute a number of commands as the nodes came online. This allowed the ability to implement changes on the master nodes without contaminating the master with a configuration management tool from the beginning of the test. These commands are executed once the instance that is started by the cloud platform is capable of executing any commands at all.

When the slave nodes were being provisioned, the bootstrapping of these nodes was much simpler than that of the configuration masters. As soon as the bootstrap scripts start, the first thing they do is create a file with the current timestamp to allow for a baseline to understand of the time the node was acknowledged as functional and capable of executing commands. Secondly the slave nodes were instructed to called the Registration API.

B. Registration API

One of the first issues uncovered in the creation of a test harness for these experiments was the concept of how to ensure as little delay between provisioning and beginning the configuration of a node happened. The key problem with this comes from the fact that regardless of the tool, there is a need to know where the master node is and to ensure access to that node is granted. This issue is actually something that a lot of the tools examined in this paper will attempt to solve within their own feature set. However, for the purposes of this study, none of these tool-specific approaches will suffice.

Functionality that requires the Master node to scan for newly added nodes is common but the limitation for this is that the scan will only run when scheduled. If this is scheduled to run on a cron, for example, every 5 minutes, any node added 30 seconds after the last scan will sit idle for 270 seconds before the next scan is initiated and at this point, the timings will already be compromised. While having the address of the configuration master hard coded, or hidden behind a DNS service to prevent dynamically allocated IP addresses from breaking the terraform provisioning scripts, it would still need to determine which configuration master to connect to. This would add significant logic in the terraform scripts and another crucial aspect of this was ensuring that any newly provisioned slave had security clearance to access the master.

Accepting that an off-the-shelf solution was not readily available for the use case of immediate recognition of a newly provisioned instance, a bespoke solution was designed as seen in Figure 5 which outlines the process used in this research. By building an API gateway that was capable of accepting REST requests from slave nodes in any of the platforms used in this research, the requirement for the ability to be accessible was met. The API gateway forwards on the request from the newly provisioned node to a custom lambda function written in Python which will take a series of parameters.

The configuration management tool passed to the lambda function will be used to find the target master node. With this instance identified, the lambda function will then connect via SSH and execute a script that references the name of

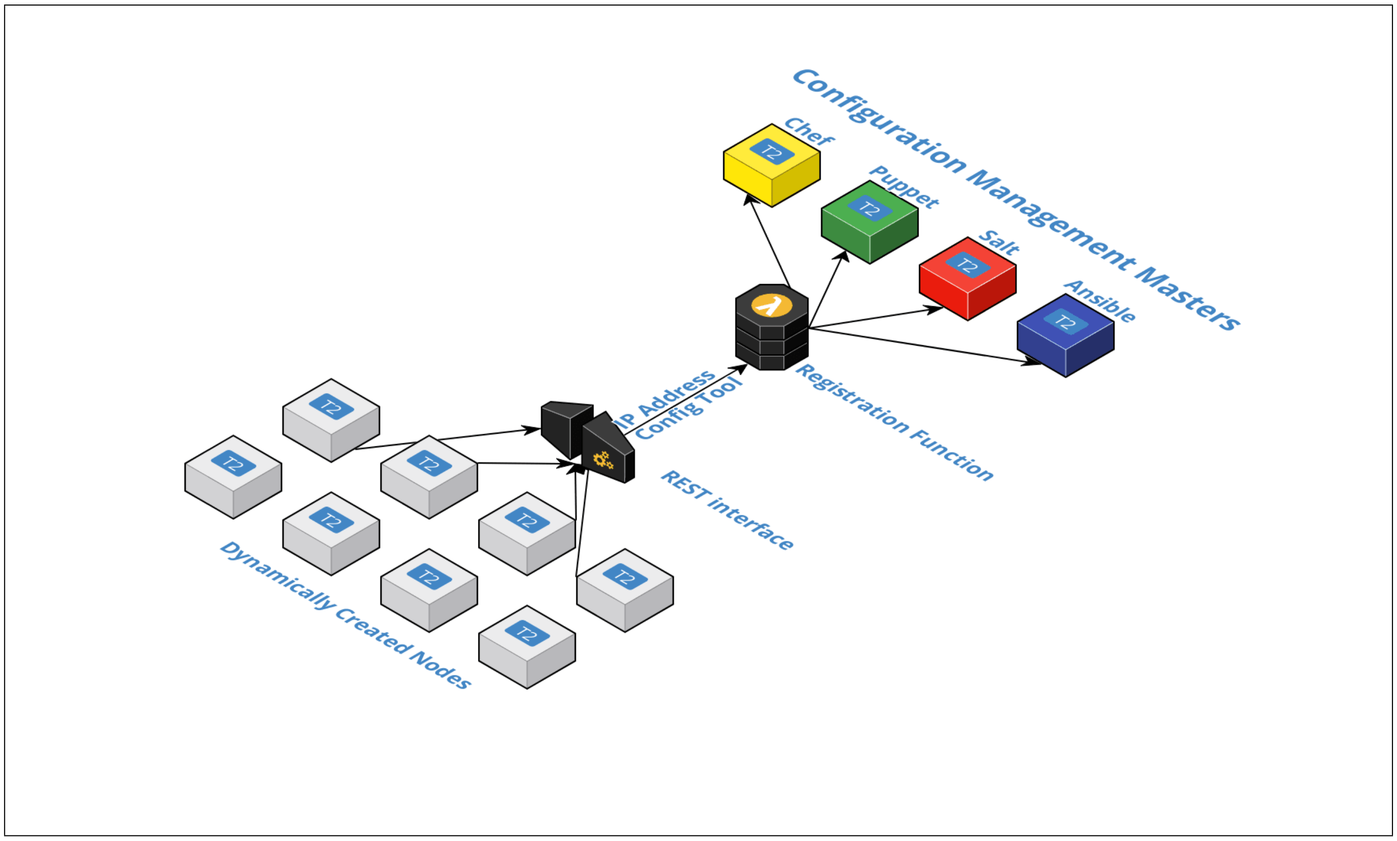


Figure 5. Architecture of Registration Process

the tool in the format of addTo¡configTool¿.sh and passes in the IP address, platform and test identifier as parameters to the script. This script is custom built for each configuration master type and is uploaded to the master as part of the bootstrapping process. The purpose of this script is to do whatever configuration is needed to ensure communication between the master and slave can occur and to trigger the tests immediately after the slave is acknowledged.

C. Bash File Library

While the terraform scripts will set up the nodes to a certain extent, to get these machines to a state where they are ready for these tests, a series of bash scripts were built to be invoked at various points in the provisioning process. These scripts allow for a level of abstraction from the terraform code keeping the the initial provisioning simple enough to prevent race conditions or timeouts. They also allow for a more parameter-focused process flow and prevent unnecessary duplication in the terraform files. Figure 6 runs through how these files take a provisioned instance and bootstraps it to be able to run the tests.

V. Results

As the overarching theme of the research is the comparison of various configuration management tools, the results are separated by test in order to see how the single activity under test differs from tool to tool. The results that have been collected over a series of executions with irregular numbers of nodes under test and across the two cloud platforms of AWS and Azure. This data is available in summarised form in Tables I, II, III, IV and V.

Table I
Time in seconds to create 10 standard Users

Tool	Averaged	Fastest time	Slowest Time	Variance
Ansible	11.04	8.22	12.95	1.78
Puppet	0.66	0.57	0.87	0.007
Salt	1.95	1.86	2.20	0.009

Table II
Time in seconds to install OpenJDK-8

Tool	Averaged	Fastest time	Slowest Time	Variance
Ansible	32	29	35	2.93
Puppet	26	24	28	2.09
Salt	189	34	293	8194

Table III
Time in seconds to create 5 folders and 50 files

Tool	Averaged	Fastest time	Slowest Time	Variance
Ansible	44.0	27.5	48.0	35.75
Puppet	0.11	0.10	0.12	0.00009
Salt	0.35	0.29	0.43	0.0015

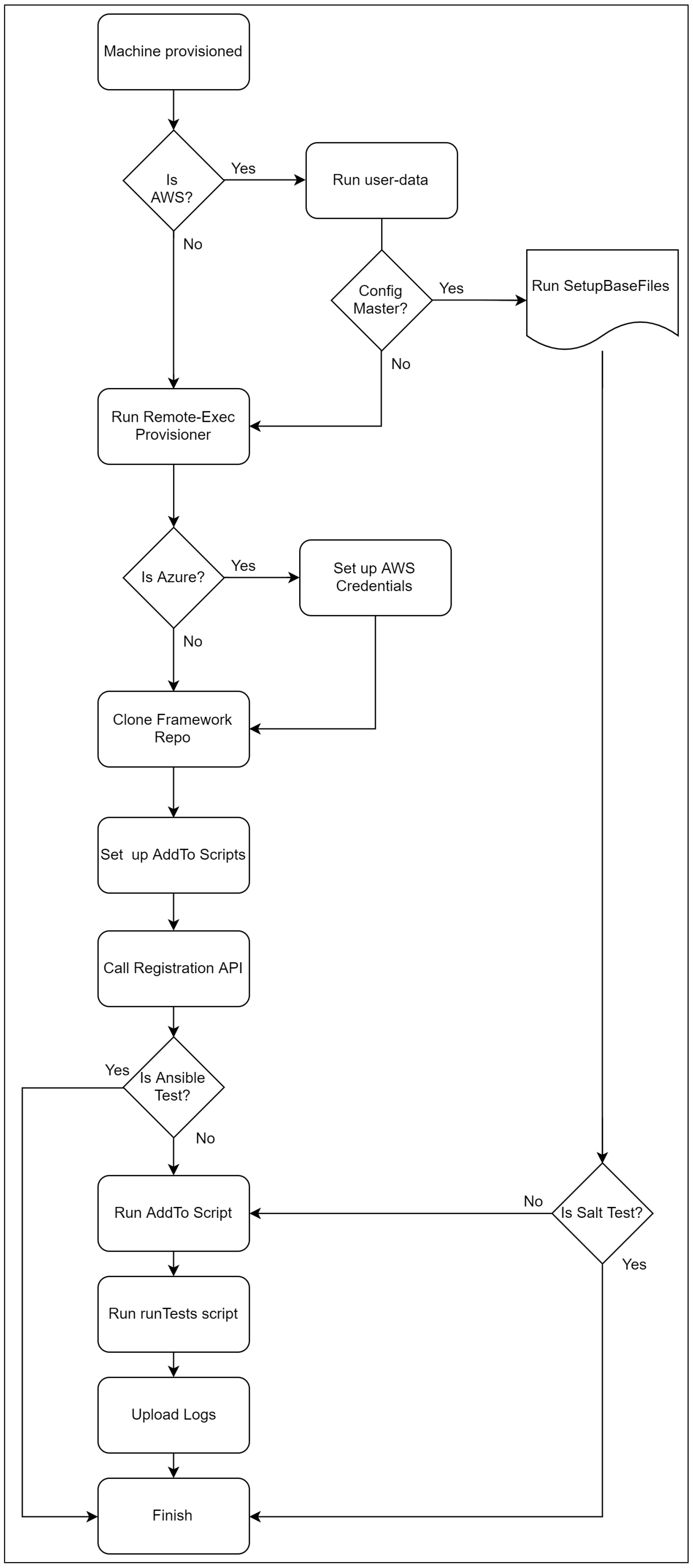


Figure 6. Process of provisioning and testing by node type

As seen in the Tables above, Puppet will consistently perform well in comparison with it's peers, only being out-paced on rare occasions but never significantly. While Ansible can perform faster on occasion, as seen with the git tests, it's slower in general for most of the tests than the other tools

Table IV
TIME IN SECONDS TO CLONE TWO REPOSITORIES

Tool	Averaged	Fastest time	Slowest Time	Variance
Ansible	96	90	100	11.22
Puppet	99	87	123	147.77
Salt	624	92	837	79686

Table V
TIME IN SECONDS TO SET UP APACHE HTTP SERVER

Tool	Averaged	Fastest time	Slowest Time	Variance
Ansible	26.4	20.1	37.6	41.58
Puppet	6.72	6.17	8.00	0.2899
Salt	36.8	10.0	69.8	553

under test. Salt is effectively acting as a wild card, showing huge potential as a high performer on occasion, achieving speeds beyond its competitors but suffering from a chronic case of inconsistency.

A. *Azure vs AWS*

Although the framework is set up to allow test execution in an automatic fashion, due to scheduling constraints, most of the tests for Azure were only run once, with a few being able to get a second execution. Those that ran twice were consistent with their earlier runs which suggests the times listed are indicative of what to expect for tool behaviour on Azure. A number of these tests were very close to the AWS timings as can been seen in Table VI, but there were some anomalies also. In particular, the installation of the JDK seemed to take significantly longer on Azure and this was run multiple times to see if this difference was reproducible. Similar increases were also seen when looking at the differences in the set up of Apache Web Server for Ansible and Puppet, both of which seeing execution times take more than twice what the average was on AWS. Surprisingly, Salt was faster initialising the webserver on Azure, but with the variance of Salt in general, there is no solid indication that it would always be faster on Azure.

Table VI
COMPARISONS OF AVERAGE TIMES ON AWS VS AZURE

Tool	Test	Average AWS	Average Azure	Variance
Ansible	Users	11.04	14.86	2.83
Ansible	JDK	32	125	1339
Ansible	Files	44	36	38.46
Ansible	Git	96	95	10.28
Ansible	Webserver	26.4	66.2	168.8
Puppet	Users	0.66	4.86	1.749
Puppet	JDK	26	65	129.5
Puppet	Files	0.11	0.12	0.00002
Puppet	Git	99	98	131.48
Puppet	Webserver	6.72	14.21	4.89
Salt	Users	1.95	5.17	0.868
Salt	JDK	189	121	7910
Salt	Files	0.35	0.44	0.002
Salt	Git	624	92	92799
Salt	Webserver	36.8	24.9	514.9

VI. Conclusions

A. Critical Analysis

Based purely on the performance metrics obtained by the test framework as seen in Figure 6.1, Puppet would be the logical choice for an organisation starting to investigate configuration management. While not the fastest at every single task, Puppet performs consistently well and if not the fastest tool at performing a task, it is usually within a few seconds of the leader. It also is the the most consistent tool, having the least variance among each run it performed.

A consideration for this report is that the framework's ability to test multiple slaves or individual nodes was expected to have minimal impact on the performance metrics gathered. Assessing Ansible and Puppet, this appears to be the case but when examining Salt, it appears to have difficulty when managing multiple slaves as the time taken to apply a state appears to be related to the number of nodes actively looking for configuration. This is not a linear connection but the trend shows that the fastest executions appeared when targeting individual slaves rather than multiple slaves, suggesting that Salt's inconsistency is potentially a concurrency issue.

Of the three tools integrated into the test framework, Puppet was the most difficult to set up and build tests for. While it never reached the level of complexity offered by Chef, Puppet was not as welcoming as Salt or Ansible for initial set up. Salt's documentation on setting up a Salt Master and associating minions with it was very clear and easy to follow. Ansible is even simpler, needing just the package to be installed on the master node through normal package manager commands to have a working configuration master. Puppet's documentation was verbose and heavily focused towards commercial users and only for the assistance of well written blogs (Kernal, 2019), was it possible to get Puppet running. This complexity is excellently represented also by the fact that the registration API will send requests to two nodes rather than one if it detects Puppet is the tool under test.

From a test writing perspective, Puppet has the benefit of in a formal repository of test cases, curated by staff and advocates of the tool for new and established users. Salt has something similar in salt-formula's (Salt, 2019) but from the experience of this study, this is not as well managed as Puppet's Forge (Puppet, 2019). Ansible's Galaxy (Ansible, 2019), is arguably unnecessary considering the ease in creating playbooks once a review of the associating documentation is complete. This suggests that if there is likely to be relatively common tasks needing automation, for any tool, there is likely an existing shared resource which is a huge benefit, however for modifying the test cases without going through one of these mechanisms, Puppet's native Dynamic Scripting Language (DSL) becomes less attractive as it's not as intuitive as the YAML structures of Salt and Ansible.

A final consideration for this research is the cost perspective required for these tools. Ansible, as lightweight as it is, was perfectly capable of running a master instance on an AWS t2.micro instance, a machine running a single virtual CPU and 1GB of memory. Puppet and Salt both required a more powerful instance, in this case a t2.medium node, which had 2 virtual CPUs and 4GB of memory. This instance type is approximately 4 times the cost of a t2.micro when looking at AWS's various pricing structures. Although Chef did not get fully integrated into the framework, it was manually set up and executed and as discovered during this exercise, Chef's documentation recommends two separate instances, one for workstation and one for master, both of which require at least the specification of a t2.medium instance which makes it at least twice as expensive as any other option proposed.

B. Limitations

The intention of this study was to target and compare the four tools identified as part of the Literature Review, Chef, Salt, Ansible and Puppet. As a consequence of scheduling and under-estimating the complexity of the tool, Chef was not integrated in time for the metric gathering activities. This constraint of time also impacted on the stability of the framework's ability to execute. While the framework is capable of running tests for Ansible, Salt and Puppet, it is not built to recover from some of the more common issues preventing a successful run such as race conditions or failed setups.

C. Future Work

To speed up testing, there are a number improvements that could be made to the framework, primarily in the reporting section. Currently, the logs are uploaded to S3 and from there, all log management is manual. This is not sustainable for large numbers of test runs as each node uploads it's own log file and these have a lot of data when only some of it is used as part of this study. A parser to gather these results, normalise and visualise them would increase the effectiveness of this framework significantly.

An obvious idea for future work is to add more tools, tests and cloud platforms to this study. Chef and Google Cloud were two proposed components of this initial research which proved to be impossible to add in schedule set for this research. It would also be useful to have the final steps of this framework full automated, with the capability of setting up a scheduler to call the required terraform files, poll the S3 bucket where the logs are uploaded to and then destroy all the resources when the logs arrive. This could have allowed for much more exhaustive testing but would have required significant investment.

References

Amazon (2019). *Amazon uptime Metrics*. URL: https://aws.amazon.com/s3/storage-classes/.

Ansible, Inc (2019). *Ansible Galaxy*. URL: https://galaxy.ansible.com/.

Basher, Mohamed (2019). "DevOps: An explorative case study on the challenges and opportunities in implementing Infrastructure as code". PhD thesis. Umeå University.

Cowie, Jon (2014). *Customizing Chef*. 1st ed. O'Reilly, p. 4.

Johari, Aayushi (2019). *Chef vs Puppet vs Ansible vs Saltstack: Which One to Choose — Edureka*. URL: https://www.edureka.co/blog/chef-vs-puppet-vs-ansible-vs-saltstack/.

Kernal, Dev (2019). *Set up Puppet Master and Agent on AWS EC2*. URL: https://www.kerneldev.com/2019/04/16/set-up-puppet-master-and-agent-on-aws-ec2-part-2/.

Klein, John and Doug Reynolds (2019). "Infrastructure as Code: Final Report". In: *Software Engineering Institute*. URL: https://resources.sei.cmu.edu/asset_files/WhitePaper/2019_019_001_539335.pdf.

Lerner, Andrew (2014). *The Cost of Downtime*. URL: https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/.

Oladapo, Victor and Godwin Onyeaso (2018). "Empirical Investigation of the Moderating Effects of Organizational Size on Ecommerce Capabilities and Organizational Performance". In: *International Journal of Economics, Business and Finance* 5.1, pp. 1–9. URL: http://www.ijebf.com/IJEBF_Vol.%205,%20No.%201,%20August%202018/Empirical%20Investigation.pdf.

Önnberg, Fredrik (2012). "Software Configuration Management: A comparison of Chef, CFEngine and Puppet". In: URL: http://www.diva-portal.org/smash/get/diva2:536382/FULLTEXT01.pdf.

Plant, Tina (2014). *Downtime costs per industry*. URL: http://ecessa.wpengine.com/wp-content/uploads/2014/08/Screen-Shot-2014-08-18-at-8.18.22-PM.png.

Puppet, Forge (2019). *Puppet Forge*. URL: https://forge.puppet.com/.

Salt, Formulas (2019). *Project Introduction - SaltStack-Formulas master documentation*. URL: https://salt-formulas.readthedocs.io/en/latest/intro/index.html.

Thomas, Delaet, Wouter Joosen, and Bart Vanbrabant (2010). "A Survey of System Configuration Tools". In: *Proceedings of the 23rd Large Installations Systems Administration (LISA) conference*, pp. 1–14. URL: https://www.usenix.org/legacy/event/lisa10/tech/full_papers/Delaet.pdf.