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

Dave Hill B.Sc.

Supervisor: Dr David White

Technology University of Dublin

In Partial Fullment of the Requirements for the degree of
M.Sc. Computer Science with DevOps
February 2021

Declaration

I hereby certify that the material, which I now submit for assessment on the programmes of study leading to the award of Master of Science, is entirely my own work and has not been taken from the work of others except to the extent that such work has been cited and acknowledged within the text of my own work. No portion of the work contained in this thesis has been submitted in support of an application for another degree or qualification to this or any other institution.

Dave Hill B.Sc.
February 2021

Acknowledgements

To my wife - none of this would have been possible without your support, this is as much yours as it is mine.
To my parents - who fought to give me as many opportunities as possible to finish this work.
To my children - who constantly kept me on my toes and reminded me how to focus and how to take breaks.
A big thanks to my supervisor, David White, who provided fantastic advice and feedback throughout the dissertation.

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. To deliver an informed strategy, an understanding of time to provision resources is crucial as it informs not just timelines for when resources will come online, but also how much they will cost. 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. This framework allows Ansible, Puppet and Salt all to run a series of tests against AWS and Azure infrastructure with analysis of the reported times.

Keywords: Infrastructure, Infrastructure as code, IaC, Terraform, AMI, AWS, Infastructure Automation, Configuration Management, Chef, Puppet, Ansible, Salt, Terraform, Azure

Table of contents

List of figures

List of tables

Chapter 1

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 tailored to the needs of a software team, 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 pushed for an ever increasing standard of rapid deployments and updates.

Configuration Management is the term used in computer science to describe the process of 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. Configuration management is also identified as one of the core tenets of the 12 factor app (Wiggins, 2017), a methodology that is identified in the industry as a guide for building, deploying and maintaining applications as a service, in particular in microservice based architectures.

This research project will attempt to utilise infrastructure automation to obtain a series of fixed configurations on an virtual machine from a selection of commercially available cloud platforms via multiple technologies with the intention of measuring the time taken to initialise the virtual machine, any associated costs of the infrastructure automation process and other consistently available metrics. To be considered complete, the initialisation process must complete configuration of any additional services called out in the experiment. The

intention is to trial this process with minimal configurations, a series of configurations for industry standard set-ups, i.e. web application servers, and pushing towards complex levels of customisation in order to see how scalable the process is. The question attempting to be answered in this case is ”What is the most efficient way, in terms of time and cost, to initialise cloud based components, and what circumstances can affect this?”

1.1 Motivation

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.

Fig. 1.1 **Downtime Costs per industry (Plant, 2014)**

Any company with an online presence will state 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). To understand the financial impact of downtime, a Gartner survey from 2014 stated the average cost per minute of downtime was $5600 (Lerner, 2014). In Figure 1.1 the estimated costs per industry based on another industry study can be seen. While this research is unlikely to see differences in terms 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, a second of downtime can cost $1800 so saving 4 or 5 seconds for every deployment can make a huge difference in financial impact. This is likely to continue to get more expensive as trends towards e-commerce and digital services continue to grow as seen in recent years with no indication of a drop in growth (Oladapo and Onyeaso, 2018). It is no surprise with these associated costs that having anything less than the optimum service available and deployed is seen as unacceptable and this is the motivation behind this study. With the results generated by these experiments, it should become evident what benefits are

available from each automation strategy, with a focus on time to deliver and cost to do so, now that a non-arbitrary figure to compare against is identified and an operational incentive to investigate is obtained.

1.2 Contribution

The intended goal for this research is to primarily highlight the identified differences in performance across the various configuration management tools used in a set of controlled circumstances. This, coupled with the secondary set of metrics of cost and usability, is intended to provide an insight into the impact of a chosen tool. This research, while controlled and unbiased, is not exhaustive. 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 the results of these experiments.

Chapter 2 will examine existing literature identified prior to this study commencing and examine industrial trends and documentation to get a better understanding of what study has been attempted previously and what findings that could be leveraged as part of this research. Chapter 3 will outline the proposed research design, discussing technologies that will be used, scope of the testing and any caveats that are taken into account. Chapter 4 will discuss how this design translated into a working implementation, with an automated test framework and a confirmation of the tests performed per tool. The results of this testing will be published and analysed in Chapter 5 and final thoughts, limitations within this research and future work will be captured in Chapter 6.

Chapter 2

Literature Review

Prior to the experimentation done as part of this research, there was a critical review of available academic literature to understand any pre-existing work in this area that may serve as a foundation for this undertaking. While the intention was to focus on areas of identical study, namely the performance and subsequent comparison of configuration management tools, the results of this literature review showed very little research in this area. As a result, this review will outline the areas deemed most relevant to this research with particular emphasis placed on studies that would prevent the experimentation from completing in this effort.

2.1 Background

While 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 a postdoc named Mark Burgess who used it to maintain the series of Unix machines he was responsible for at the time (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 configuration management 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.

Acknowledged that the traditional use of shell scripts to manage infrastructure once deployed, due to it simplistic nature, was far more error prone and likely to result in configuration drift

(O'Connor *et al.*, 2017). This is a state where multiple machines intended to be identically configured end up differing due to manual intervention for reasons such as patching individual machines and troubleshooting problematic instances. Configuration management is designed to become a mechanism to allow robust and reliable provisioning of machines as needed. As noted by Perera and Beck (2018), it was an accepted practice in many Information Technology companies to roll out a deployment via a series of documented manual steps. As well as being a constant source of strife typically between development and operational teams, this was also frequently subject to human error which could cost companies significantly in lost opportunity cost as well as manpower effort from the teams deployed to debug and resolve any issues as a result.

With all these acknowledged issues with manual deployments, the business case for automated configuration management is an easy one to sell. This is only further compounded by public use cases, such as when an issue that occurred in an installation for one of the systems used by the New York Stock Exchange, it was publicly credited that the configuration management system was the reason for the rapid reversion of the software in question and allowed trading to resume in under 90 minutes (Agarwal *et al.*, 2018). The relatively recent shifts towards massively scaled infrastructure to enable services such as Netflix and Youtube, online video streaming giants who can claim to be responsible for over a quarter of all internet traffic (Sandvine, 2018), and therefore can expect a consistently high demand for access to their services, would be incredibly difficult without configuration management tools. An additional level of complexity has now emerged as services which do not have a consistent demand, such as Amazon's e-commerce site during Black Friday sales, are focusing on an ability to scale rapidly to address spikes in user connections or usage trends to optimise infrastructure costs. This rapid scaling would be impossible to match with manual deployments and further validates the need, not just for configuration management as a concept, but for performant, reliable and flexible configuration management in these industries.

With the push towards cloud computing and the derived concept of Infrastructure as a Service (IaaS), the ability to provision and configure hardware became critical to take advantage of the potential financial savings offered by IaaS. To utilise these services efficiently, IaaS providers typically offer API, CLI or third party library access, for example the boto3 library offered by AWS. This crucial gateway to these platforms enabled the Infrastructure as Code (IaC) movement which, as seen in Figure 2.1, derives concepts from the Agile manifesto and the DevOps movement but is not limited to them, as it's scope is not tied exclusively to application development as these methodologies typically focus on. Instead, IaC has traditionally focused on the two key areas highlighted by the potential of IaaS, the first being

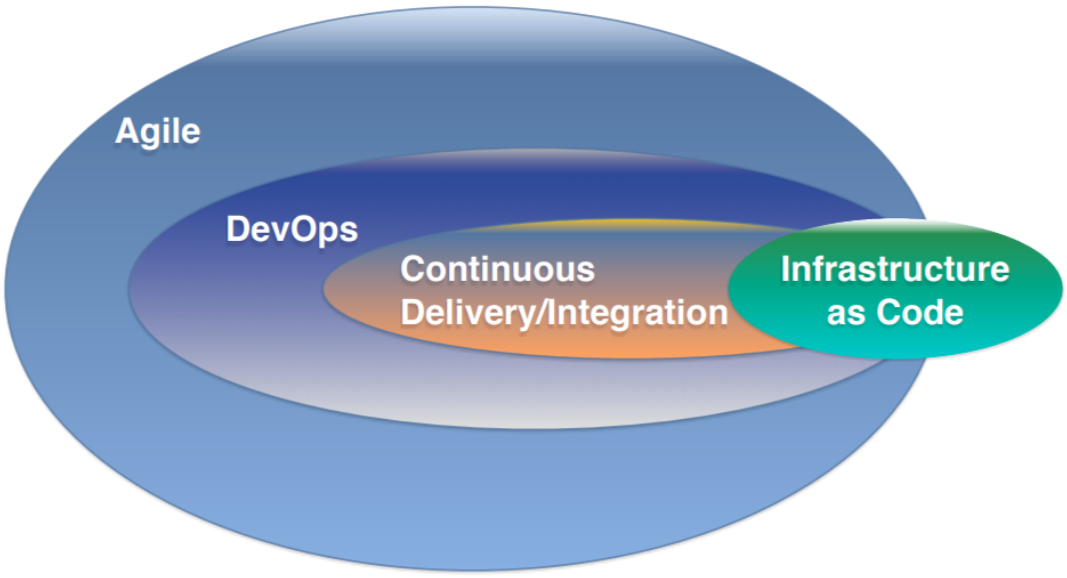

Fig. 2.1 **Relationship of IaC to Agile and DevOps (Klein and Reynolds, 2019)**

the provisioning of infrastructure as needed, and the second being the configuration of this newly provisioned infrastructure, which is the key area that will be investigated here.

IaC has not been without it's challenges in adoption throughout the IT industry. Typically needing far less boilerplate code than standard application development, IaC as a result has been frequently underestimated in it's complexity and potential to cause issues. In their studies, Schwarz *et al.* (2018) and van der Bent *et al.* (2018) focused on "Code Smells" in IaC, based on existing research done on the puppet tool initially. The team validated that a level of quality can be derived from code smells detected on puppet implementation files and that many of these were technology agnostic, as proven with their experimentation of implementing similar checks on Chef. These code smells, a concept used traditionally by static code analysers, are indicative of poor practices within the code being examined. In software development use cases, these code smells are used to ensure that best practices and style guides are implemented throughout the code produced by teams and these have always been traditionally open to customisation to allow for peculiarities from each team using them to be baked into their validations. It is through this ability to manipulate the checks performed that the team were able to apply these checks to Chef also, an effort which is not only encouraging to see as IaC becomes more recognised as a discipline, but also more importantly, indicates that there are identified poor practices in IaC which should be avoided.

As noted in their future work, this movement is still in its infancy so while it is worth noting what criteria the code sniffing software was aiming to identify, there are no mature or industry standard tools to use for this research. With this in mind, no formal validation on the IaC code will be done, as many of the languages in this exercise are not yet supported by these recent studies into static code analysis for IaC.

A common theme for most of the configuration management tools available, and certainly the ones selected for this study is their syntax. Each of these tools use structured data such as JSON or YAML to build instructions for their engines to allow quick and efficient script creation. Using this file based approach, configuration can be checked into source code management tools like git or SVN, just like any other source code or file based assets for development teams, enabling change management, versioning and rollbacks of the files to previous states should the need arise. This structure is not as complex as the source code used in popular programming languages such as Java or Python, offering a lower barrier to entry and encouraging engineers, unfamiliar with programming syntax, to contribute to their maintenance. This simpler implementation has allowed for some experimentation with further abstraction of the configuration management process as seen in the study performed by Klein and Reynolds (2019), potentially leading to higher adoption of IaC tooling.

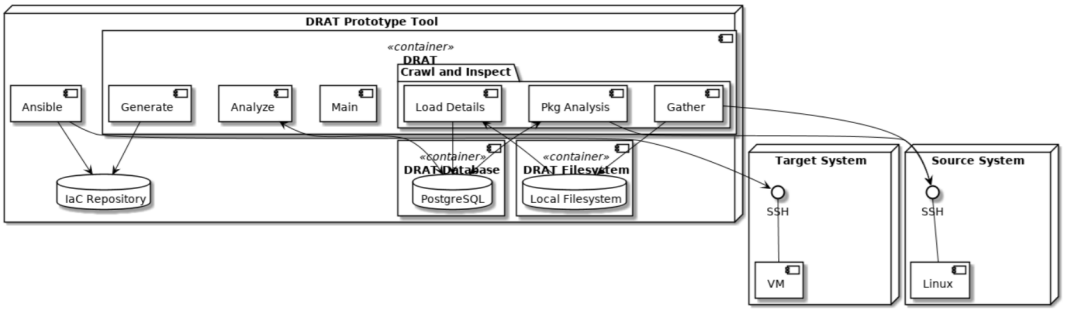

Fig. 2.2 **Process Flow for DRAT tool (Klein and Reynolds, 2019)**

With their prototyping tool, DRAT (Deployment Recovery and Automation Technology), the team, using a series of python scripts, developed the capability to automatically generate configuration management scripts for an existing infrastructure, albeit limited to a narrow scope of technologies initially supported. The process flow used by the DRAT proof of concept can be seen in Figure 2.2. This tool would begin it's construction by first scraping the data from each machine's package manager to understand what libraries were installed on each machine. From here, it also searched through a series of files paths to grab associated configuration files corresponding to each of these libraries. This information was all returned back to a centralised parsing node which would collate the information and attempt to put

some rules in place for the purposes of summarising the changes and making the most efficient determinations of what would be needed to recreate this infrastructure set up. This included rules such as checking for duplicates across the nodes to understand if generic rules were needed or learning which libraries were installed as dependencies to understand which configuration files could be ignored. Once this analysis was done, the tool would then automatically generate Ansible artifacts according to the tools best practices with the intention of establishing a foundation for systems not currently using any configuration management options. While acknowledged as something that, to scale up for industrial use, would require considerable effort and maintenance, the concept itself is not one that is inherently complex, instead being more time consuming that anything else. This kind of work will likely lend itself well to a form of AI or machine learning in the future.

2.2 Benchmarking

The most closely associated element of academic literature that could be identified for this research was that of a tutorial paper outlining the process used to perform benchmarking of IaaS using a tool called Cloud WorkBench (Scheuner and Leitner, 2019), published only a few months earlier. This paper champions a similar concept that this paper sets out to achieve, a desire to understand the performance and capabilities of IaaS tools, but differs in that the targeted set of tools are the cloud computing platforms, such as Amazon Web Services (AWS) or Microsoft Azure, themselves.

In order to do this, the team set out to build a utility that would automate the provisioning of virtual machines on these platforms using proprietary APIs, then configuring them via the more popular configuration management tools. It then leverages the existence of Chef dependencies to perform configurable benchmarking activities, ranging in expected duration from seconds to hours. The results of these experiments are gathered and returned back and then using the initial creation tools, the Cloud WorkBench tool will destroy or release the machines, with the results now freely available to be examined and interpreted.

While this does not align with this research entirely, thanks to the research team making the tool available for review (https://github.com/sealuzh/cloud-workbench), there is an opportunity to investigate the usage of the benchmarking functionality as well as understanding how the team accurately gather metrics from the various providers. The tool itself is written in Ruby and utilises it's Rails framework to generate a web based front end. The choice to use Ruby makes even more sense when realising that both Vagrant, the tool used for provisioning the resources, and Chef, are both written in Ruby themselves, and this synchronisation should

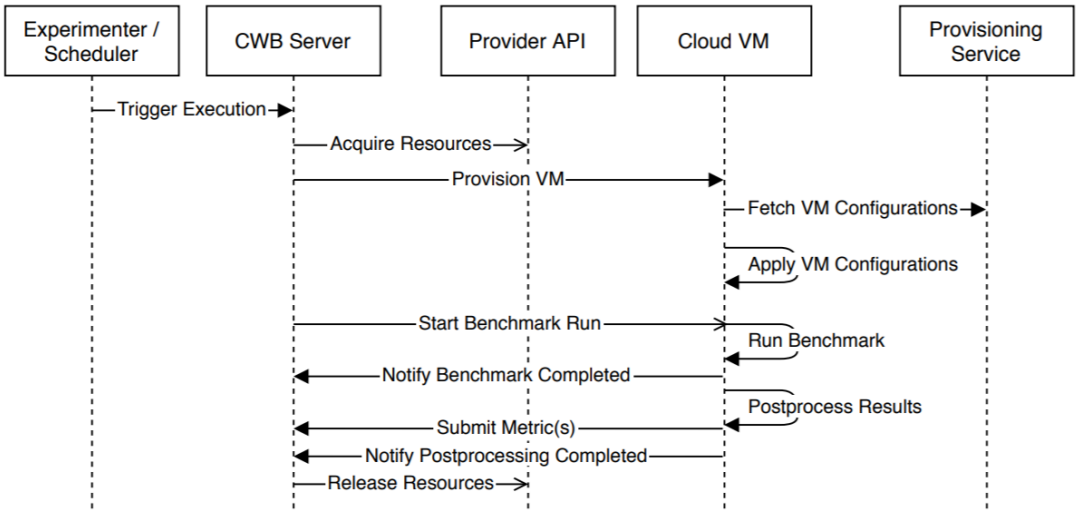

Fig. 2.3 **Process Flow for CloudWork Bench tool (Scheuner and Leitner, 2019)**

simplify or remove completely any integration issues between each of the components. The concept of using tools that will be easy to integrate extends towards the scheduling mechanism, which simply uses a cron to establish the frequency of the benchmarking activities. The team behind this study identified that the results derived from the benchmarking exercises are not immutable. As the IaaS platforms evolve and grow, the underlying hardware can change and this can affect our results. Depending on the time of day, demand for resources, especially on Virtual Machines on shared physical hardware, can affect the results of this benchmarking also. In order to attempt to mitigate misleading results, these measuring exercises should not be singular executions. By scheduling the experiments to run over extended periods of time, at multiple points in the average day to take timezone usage into account, as well as tracking the impacts of different times during the week and even during the month, if schedule allows, not only offers a more realistic metric that can be averaged across the entire duration of the testing, but this also allows us as researchers to infer usage trends on these IaaS providers and extract more meaningful data to understand *why* we're seeing these differences in performance.

As Cloud WorkBench is an open source tool, with a concise and polished GUI, there is potential for us to re-use this reporting mechanism to offer a streamlined reporting function. Part of the research done for this project involved reviewing potential candidates for cannibalisation for our needs.

2.3 Comparison of Tools

The core search definition used for this project was around the comparison of Configuration Management tools and derivatives of the same, focused primarily on individual tool performance speeds. The initial collection of academic research was focused on any research published in the previous eighteen months in an effort to focus on the most relevant and up-to-date academic research. This initial set of search parameters lead to a research paper dedicated to the cataloguing of study into IaC which provided a level of insight towards what could expect to be found in this research. In their systemic mapping of this area of study Rahman *et al.* (2019), with generic search terms such as "infrastructure as code", "devops" and even some tool names to afford a more targeted approach, the team identified over 31,000 publications, which is promising. However, in their analysis and by applying a stricter set of search criteria within this initial set of results, the team ended up with only 32 publications.

The team then categorised these into four distinct groups, articles related to new frameworks or tools relating to IaC itself, the rate of adoption of IaC across organisations via case studies and surveys, empirical studies focused on analysis of investigative questions regarding IaC and finally, the category that most relevant to this study, the testing of IaC as a concept and the associated artefacts. However, even this targeted area has focused on concepts that do not relate to this research, instead, half focusing on the idempotence testing and test case reduction and the others examining the use of automated testing of configuration management scripts to ensure correct behaviour.

Throughout the paper, Rahman *et al.* discuss the limitations of their efforts, recognising that their initial sources were five academic databases and this could easily allow the omission of other publications. Also, their exclusion criteria, while some are easily justifiable, such as duplicated papers across the databases or work that has not been peer reviewed, can be excessively restrictive in other areas, most notably, their initial selection criteria within the original set of results which demanded that the literature have cited one of four books, or recursively been cited by another publication that cited one of these selected books either. While these books are logical reference points for anyone investigating configuration management, this can obviously fail to include many publications that did not determine a need to cite such literature. This exclusion criteria could also be responsible for the identified lack of research by Rahman *et al.* for areas such as test coverage, test practices and testing techniques. Our intention to perform speed of execution tests is acknowledged to be a niche domain within these research groups.

In an attempt to be as thorough as possible with this research, the initial restriction of publications only submitted recently was removed to see if there were any publications that compared the configuration management tools on the market at an earlier point in time. This broadening of the search terms did not serve to drastically grow the number of related studies, which, when considering the relative youth, not just of configuration management as a concept but the fact that there would be competitors and IaaS platforms to compare is still something that is only widely available in the last decade or so. However, an interesting survey of system configuration management tools was identified which took 11 configuration management tools from a decade ago with the intentions of comparing the tools across multiple criteria (Thomas *et al.*, 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

Fig. 2.4 **Tools compared by (Thomas *et al.*, 2010)**

As seen in Figure 2.4, many of the tools used in the study by Thomas *et al.* 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 to get a series of results for each comparison point and would then assign tools to these result buckets in an effort to grade them in these reviews. For example, when examining the level of community engagement in each of the tools, the bespoke grades were large active communities, small active communities, small scattered communities, inactive small communities and no communities of note. Unsurprisingly, the open source tools here were the ones with large active communities and as the tools moved towards commercial and obscure, their grading dropped through the ranks. This is just one example of the criteria compared, the study went on to also compare the tools across 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 a strong likelihood 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. However, their acknowledgements of feedback from users on these gradings suggest that dedicated users of these tools may have also fed into the assessment of these tools. This study, while informative, is unable to provide any foundation for the research this paper is targeting.

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 went through installation steps and dependencies of each of the tools, as well as some of the attributes covered by Thomas *et al.*, such as documentation and community. The focus was definitely narrowed to focus with greater clarity on the technical details of these tools with observations around the release package format as well as alignment with the installation documentation but this too 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.

2.4 Summary

Configuration management, as a recognised, significant component of IaC, and similarly critical to IaaS, is something that has had many facets of it's existence reviewed and scrutinised. Considering it's operational impacts on any IT infrastructure, and explosion in popularity in recent years, there is clearly an abundance of literature referencing the high level concepts around configuration management. As noted by Rahman *et al.* in their study, there are thousands of published artefacts that can be associated with configuration management. And this popularity is expected as the methodology, techniques and tools have vast scope within each of them from an analysis point of view, but incredibly, considering the old adage that time is money, it's quite unusual to be able to find any examples of studies in the same area as this paper, namely, an experiment to determine an indicative level of performance for a popular set of configuration management tools.

What can derived from this review of available literature is that a significant amount of the research reviewed was focused on case studies and surveys of configuration management in action across multiple industries. This is evidently something that commands significant interest in both academic and commercial communities and taking into account some of the learnings from this literature review, should prevent identified obstacles and issues from occuring in this research while also identifying potential headstarts that can be utilised from a technical perspective.

Chapter 3

Research Design

Chapter 1 introduced the core aim of this research, namely the investigation of the impact of tool choice on the performance and cost of Configuration Management. Chapter 2 was designed and went on to provide a review of research into configuration management with a specific focus on aspects related to performance, usability and capability, with a particular focus on the more recent migration of industry IT infrastructure moving to cloud platforms. This chapter will outline the approach for performance testing for this research and provides details of the test harness created to perform this testing.

The foundation of this research is to compare four commercial tools used for configuration management across a series of controlled experiments in order establish comparative metrics on performance of these experiments as well as a review on the complexity and ease of use of each of the tools chosen. 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 selected configuration tools will have, the decision was made to 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 particular tool over another. While this is not expected, the basic infrastructure offered by these cloud providers is very similar and as such, any significant deviations between the performance of the same configuration management 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.

3.1 Configuration Management Technical Concepts

While the deliverable behind each of the configuration management tools of this research is the same, namely enforcing a desired state onto a new machine, the implementation of these tools is not uniform, in either language or logic. In order to recognise some of the challenges in this research, as well as identifying potential reasons behind differences in performance, it is crucial to understand some of the key architectural concepts behind the tools being researched and how they differ among themselves

3.1.1 Master/Slave Topology

A relatively consistent element of configuration management systems is the presence of a "master" node which is traditionally responsible for being the source of truth for any machines that require configuration beyond the initial provisioning, which are known as "slaves" (or "minions" when discussing Salt). With the presence of a master node, this is the machine that is responsible for determining what configuration is available for a new machine to utilise, and for more mature infrastructure set-ups that have configuration files available to enforce these desired states, the master node is typically the only agent with access to these files at source. As per the heavily advocated 12 factor app (Wiggins, 2017), any configuration management artefacts, regardless if they are scripts or dedicated input files for one of these tools, they should be stored in a source control repository using something like git or SVN to handle their change management and ensure any changes can be reversed if necessary.

The connection between the master and the storage repository of these configuration management files removes the need to enable access for each of the slave nodes to the source code repository. This should in turn simplify the infrastructure needed to manage the slave nodes but obviously the need to connect to the master node has it's own challenges that will be explored later in Chapter 4. The instructions embedded within the input files for each of these tools are intended to be a simplistic rendering of what the relevant command would be if entered manually. This is done for ease of review and maintenance as well as to encourage adoption. It is entirely possible that for engineers who invest heavily in one of these tools at the beginning of their careers to underestimate the complexity of some of the tasks these

tools automate through the use of a syntax focused on usability. As the tool ingests these files to understand the state the target machine should be in, depending on the tool in use, they will either send commands that interpret these files and remotely execute the commands within them, taking into account any logic within the files such as conditional statements, or look transfer the configuration files to the target machine for execution by a local daemon.

It is worth noting that of the tools named as part of this research, all of the tools bar Ansible market solutions that do not require a master node at all, instead having all of the configuration management tool's dependencies installed on the target machine. While theoretically, there is nothing preventing any of these tools running in an isolated fashion like this where each node gets a full install and master style configuration, this is not an efficient set up for enterprise-level IT infrastructure and therefore will not be focused on within this research.

3.1.2 Push vs Pull delivery

Another key facet of configuration management tools is understanding how the slaves communicate with the master node to obtain their configuration and register their existence. It is a common feature of these tools to track each of the slave nodes that have registered their existence to the master and this is usually built upon with these tools to offer analytical insights or dashboards into the health and status of each of these slaves from the master's perspective. The fact that these metrics are tracked from the master's perspective is important to highlight. If a slave has it's local agent fail or if there is an external alteration to prevent the slave and master communicating, such as a security group change in AWS, then the master will report this slave as unavailable and may flag it's status for investigation. In the meantime, the slave can continue to function exactly as desired from an application perspective.

As seen in Figure 3.1, there are two delivery models to be understood. The first is the Push model which is primarily driven by the master node. In the push model, it is the responsibility of the master node to ensure that any slaves connected to it obtain their configuration. Once the master node is aware of any slave nodes, it will determine what configuration the new machine needs and it will initiate the execution of the configuration management. These commands are executed and the status of the execution is reported back to the master node also. Using this mechanism, the master node tracks the status of each node and will push out updates as needed, retry failed executions and can report on various metrics across the family of nodes it manages. Of the tools that will be examined, Salt is the main example of a push delivery model, although Ansible has similar features.

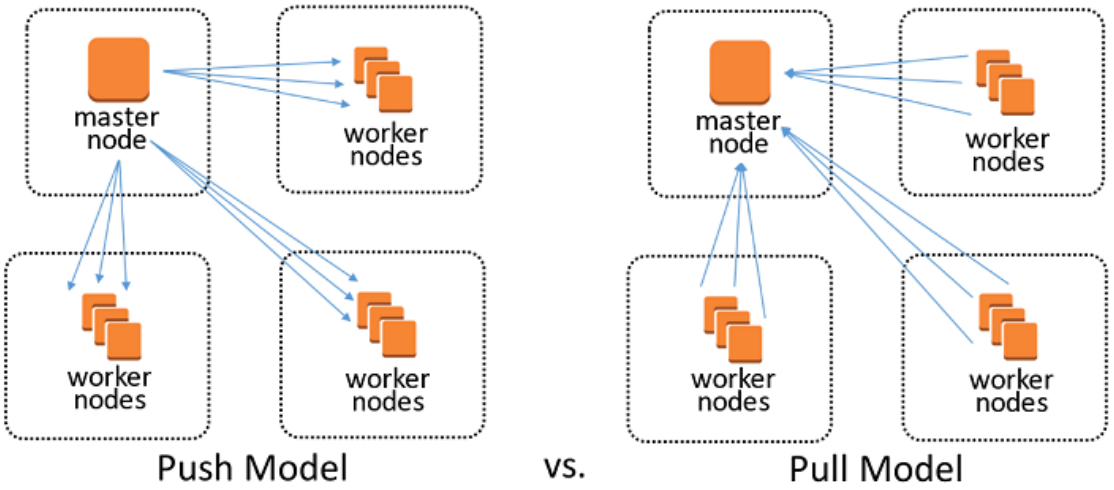

Fig. 3.1 **Visual representation of push and pull delivery models (Amazon, 2019)**

The pull model has a much simpler role for the master node. In this instance, any newly created machines are usually bootstrapped with the configuration management daemons with the location of the master node baked into the provisioning. As a result the newly formed node will go to the master and based on the information it sends to the master, it will request the configuration it should be implementing. In a stark difference to the push model, here the master is not tasked with maintaining details of each of it's nodes and their current state, instead the onus is on the slave node to query periodically the master node to check for relevant updates to it's desired state as well as ensuring that it has everything it needs. Puppet and Chef would both be examples of tools using the pull delivery model.

In practical terms, the difference here is what initiates change. In the push model, if there is a change to a configuration needed across the infrastructure it manages, the master can be notified by a change in the source control repository it is affiliated with and from there it can start to roll out the changes to every node it has a record of and track when the state has arrived at it's desired form. In the pull model, to pick up new configuration would require waiting for the next heartbeat request, a call made by the slave node to the master to check for changes, or possibly triggering a request for configuration updates manually.

3.2 Planned Approach

The intended approach for this research is to capture specifically the performance of the configuration management tools in establishing how to get a standard machine in it's desired state. As such the process of actually provisioning infrastructure, the speed of which is

not a metric that will be tracked or reporting on, is critically important in order to perform the experiments. Once the machines are up and running, 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.

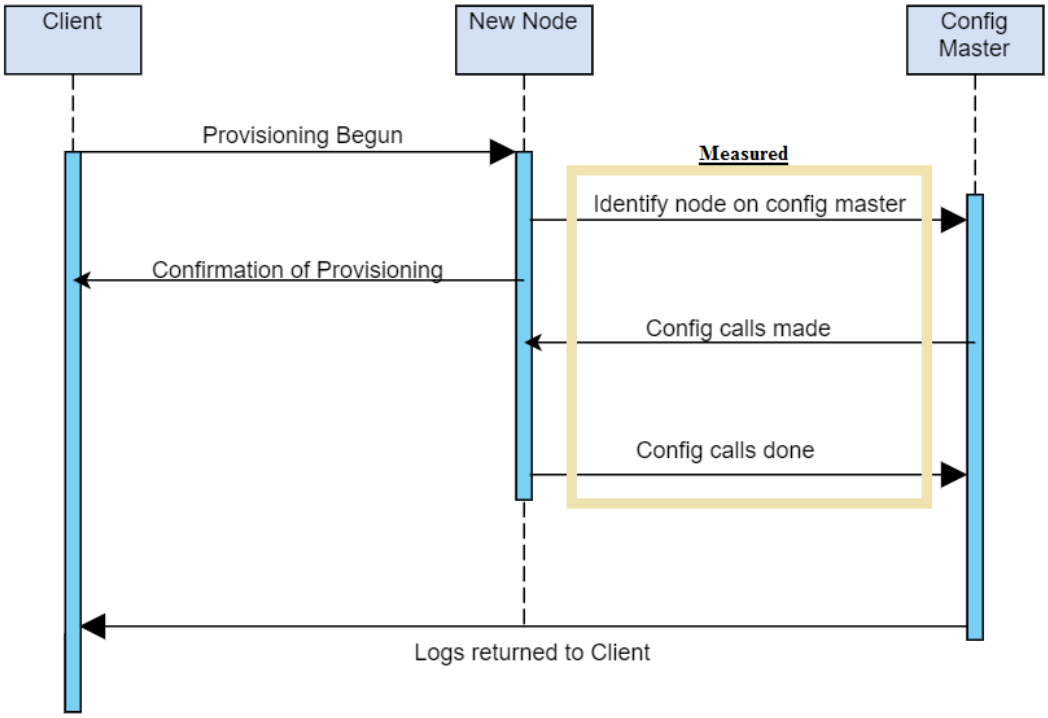

Fig. 3.2 **Provisioning steps targeted as part of this study**

Once the configuration has completed on the node, a log will be used to obtain the time needed to execute the configuration management and these logs will then be collected for analysis. As seen in Figure 3.2, only tracking the time taken to configure the node into the desired state is explicitly captured, provisioning the node from the cloud provider is not considered part of the timings, but once the node is declared provisioned and available by the cloud provider, this is the point where the timings will begin. Once the configuration is successfully applied *and* reported back to the master is when the timing will stop. Time to collect the logs will not be considered any time taken to collect the logs as relevant for this research.

As an aside to this research, there will also be an investigation into the usability and ease of installation for each of the tools. As per the core research topic, getting a system configured

as quickly as possible is the true objective. If a tool is marginally faster to execute but the learning curve is much higher and integration was much more difficult, this additional complexity is worth noting before adopting any findings based off of this study.

3.2.1 Test Cases

In order to accurately compare the tools, a consistent set of criteria is needed to be executed against each tool and on each cloud platform. These tests were defined with an end state in mind rather than a concrete implementation. The reason for this is due to the differences in syntax and functionality and as such only success criteria can be defined, rather than milestones within the tests. For these tests, the target machine will be running the popular Ubuntu flavour of Linux. The machines will all be running version 16.04 (codename Xenial Xerus) as it's well established as a stable version of this version of Linux, it is listed as a supported OS for each tool and it has existed long enough to cultivate a community that may have dealt with common issues while attempting setup.

		AWS			Azure			Google		
Tool	Test Cases	1 Machine	5 Machines	20 Machines	1 Machine	5 Machines	20 Machines	1 Machine	5 Machines	20 Machines
Puppet	Initial Set up with user login									
Puppet	Installing JDK/Python version									
Puppet	Creating directory structure and files									
Puppet	Clone repo									
Puppet	spin up webservice									
Puppet	restart and validate									
Chef	Initial Set up with user login									
Chef	Installing JDK/Python version									
Chef	Creating directory structure and files									
Chef	Clone repo									
Chef	spin up webservice									
Chef	restart and validate									
Salt	Initial Set up with user login									
Salt	Installing JDK/Python version									
Salt	Creating directory structure and files									
Salt	Clone repo									
Salt	spin up webservice									
Salt	restart and validate									
Ansible	Initial Set up with user login									
Ansible	Installing JDK/Python version									
Ansible	Creating directory structure and files									
Ansible	Clone repo									
Ansible	spin up webservice									
Ansible	restart and validate									

Fig. 3.3 **Provisional collection of Test Cases**

Figure 3.3 shows the list of initial test cases to be performed as part of this research. In each of these tests, the test log will obtain, in milliseconds if possible, the time taken to implement these changes and report back to the master. Any findings that are relevant but not these explicit metrics will be collected and discussed as part of the conclusions where a more holistic view of these configuration tools is also dissected. As these findings are based off qualitative research rather than quantitative, only the findings deemed most relevant to the research will be included in this report.

Part of the reason for the investment in the test harness supporting this research is due to the need to run several iterations of these tests on each of the cloud platforms and at varying intervals throughout the day and week. This should allow for any potential spikes in activity to be identified and a more balanced average obtained. A further investment in tests designed to stress each of the configuration management tools is also a possibility in order to develop more meaningful metrics if needed.

3.3 Components

As the execution of this framework requires some orchestration and set up, this section will outline the core components of the test framework itself and how they were constructed.

3.3.1 Provisioning

As alluded in a previous section, the creation of the nodes for this exercise is a vital part of the experiment and something that requires significant automation. In order to perform these experiments, there is a need for virtual machines to act as the master nodes for each of the configuration management tools as well the ability to provision empty template nodes at scale. Cost for this study was also a concern, as having all these nodes kept online for testing would have not just been costly for their basic presence, but a further necessity of wiping any configuration from these nodes would then have also been necessary. After evaluation of provisioning tools on the market, Terraform was selected for it's cross platform feature set as well as integrating tightly with most of the configuration options available as part of automated instance creation.

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. Ironically this is a task ideally suited to configuration management systems, but 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.

The benefits of this were two-fold. Firstly 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. Secondly, this bootstrapping

works as though it is executing commands on the terminal itself, which allows for greater freedom albeit at the cost of verbosity when compared to a configuration management tool.

Because of the flexibility of Terraform's syntax, the scripts are capable of ingesting variables at runtime, and this flexibility is what allowed the capability of provisioning nodes in greater numbers for the experiments within this research. 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.

3.3.2 Registration

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. For single runs, this issue is not too problematic but for this testing to occur where multiple nodes would provision at once, across multiple cloud providers, with potentially having the configuration masters themselves assigned dynamic addresses, automating this process would prove difficult.

This issue is actually something that a lot of the tools examined in this paper will attempt to solve within their own feature set. Ansible for example champions a process called Dynamic Inventory which, depending on the provider being used, will attempt to scan through the available network for any nodes not registered in it's hosts file and add them to it if they meet the criteria passed to the dynamic inventory. Salt, instead requires that you bootstrap the minion with the details of the salt-master in it's bootstrapping and will then start to request it's configuration. These are just some examples of the solutions offered by these tools, and are well documented. However, for the purposes of this study, none of these 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 of 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 investigating the viability of this option, it was determined that trying to run this script more than once every 30 seconds where the possibility of new

nodes being identified would cause issues on the master node itself and even if this was optimised further, any artificial delay at all would still impact the findings of the research and pollute the metrics beyond redemption.

A further investigation into the baking of additional information in the slave nodes to allow for automatic registration also revealed further issues with this potential solution also. While having the address of the configuration master could be 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. While implementing security is not strictly required as part of the experiments being implemented here, even if security concerns were entirely ignored, having the capability to connect to another host would require at the very least a series of modified security files on both the configuration master and the slave node which would leave the nodes exposed to the internet and, therefore, potentially capable of being tampered with. This risk was seen as unacceptable and while an investigation into platform specific rules was done, the limitations this would impose in the research, when using multiple cloud providers to connect to a single configuration master in one provider, was also significant enough to warrant investigation elsewhere.

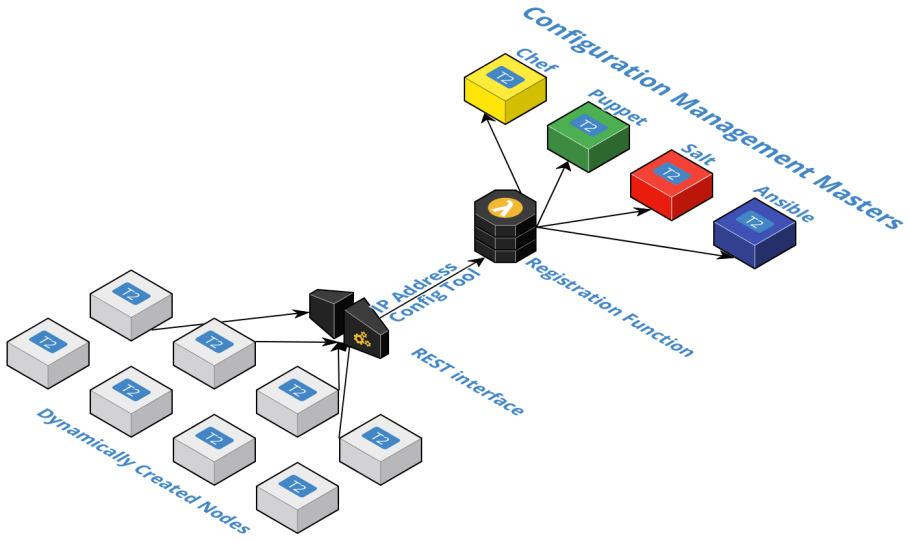

Fig. 3.4 **Architecture of Registration Process**

Accepting that an off-the-shelf solution was not readily available for the use case of immediate recognition of a newly provisioned instance, capable of dynamically identifying the configuration master needed and accessible from multiple cloud providers, a bespoke solution was designed as seen in Figure 3.4 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 itself forwards on the request from the newly provisioned node to a custom built lambda function written in Python. This function will only run if two parameters are passed to the function, the IP address of the node making the request and the configuration management tool that is required to be registered under.

The configuration management tool passed to the lambda function will be used to look through the EC2 instances of the account for a tag that matches the name sent to the function. With this instance identified, the lambda function will then connect to that function over SSH and execute a script that references the name of the tool in the format of addTo<configTool>.sh and passes in the IP address as a parameter 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.

3.3.3 Log Collection

In order to obtain a clear view on each action as part of these experiments, the scripts and commands used to control the bootstrapping, co-ordination, execution and capture are all modified to log their actions with a timestamp to a centralised log which, as part of the final steps in the logging process, will mark the file with identifiers and timestamps before uploading them to S3.

3.3.4 Log Analysis

These logs are stored in simple text format for now and while theoretically, it would be possible to automatically parse the timestamps generated by the commands and scripts constructed as part of the test harness, each command that is executed by the tools will have their own individual style and syntax. Rather than investing in a generic log parser to be able to strip out the relevant information for each tool and test, this time can be better spent in

investing in the automation of the test execution instead. While this ensures arduous manual activity when reviewing the test results, it should also reduce any potential errors brought on by an attempt to parse logs in a variety of formats.

3.3.5 Summary

This design is a record of the planned research for this series of experiments. While every effort will be made to adhere to this plan to ensure recreation and peer review, modifications that need to be made to achieve the intention of the research, namely the performance testing of each of the configuration management tools, will be done and documented in the discussion of the results and in the conclusion. All of the source code used for these experiments are also available on gitHub for review.

Repository Details

https://github.com/RedXIV2/terraform

Chapter 4

Implementation

In the previous chapter, the approach for this research was designed and outlined at a conceptual level to achieve the goals called out by this research proposal. While there was a significant attempt to establish the low level detail in this design, throughout the course of the research, there were a number of modifications needed to the proposed design based on technical and schedule limitations.

4.1 Test Framework

Although this research is targeted at widely available tools and the measurement of their performance and usability, a significant amount of work went into the test harness used to perform this research. This harness was built in order to allow repeatability in the execution of these tests as well as automating any of the manual steps needed to prepare the test environment to ensure any time needed for setup was reduced as much as possible. It does this by utilising Terraform as the key tool for provisioning, both the configuration master and any slave nodes as part of the test. As part of the Terraform, the nodes are instructed to clone the code for the framework onto it and depending on the role of the machine being provisioned, a series of scripts will be run which will trigger the tests before uploading the results. As the core components of this framework are examined in the next few sections, it will be highlighted why decisions on implementation were made, any considerations that were made in deviating from the initial design, as well as detailing the functionality that each component offers and it's relation to the overall project.

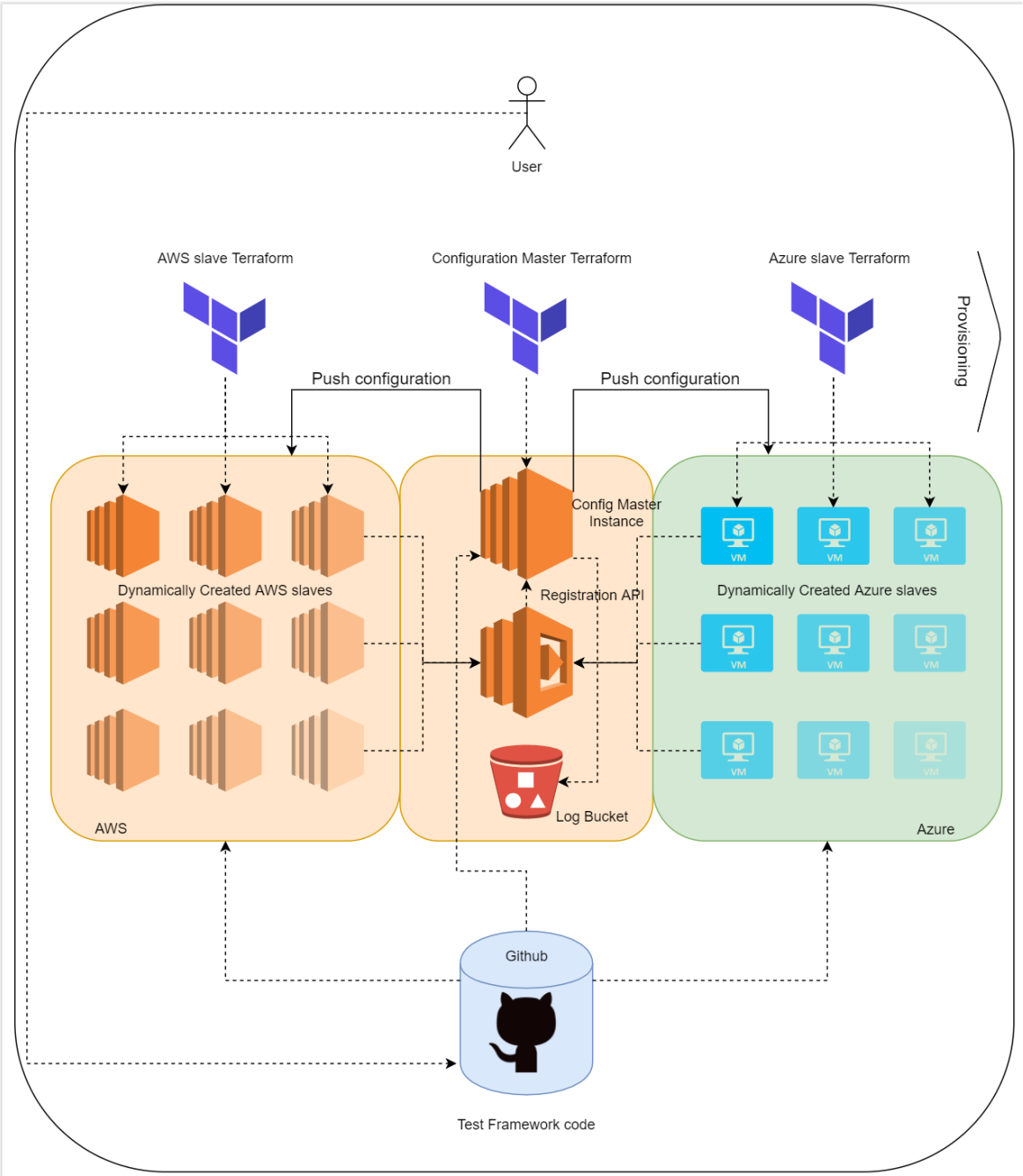

Fig. 4.1 **Test Framework**

Figure 4.1 outlines a high level overview of this framework which should aid in understanding the various components and their communication paths with each other. It is worth noting in this diagram that S3 bucket used for logs, the registration API and the configuration master node will always be hosted in the AWS environment but the dynamically created nodes are capable of being spun up in either cloud platform, AWS or Azure.

4.1.1 Terraform

The provisioning of new machines for testing was identified as being critical to this project. There are three main scripts we examine in how this provisioning process was implemented, one for the configuration master machine, and one for both AWS and Azure to launch test machines. While the ability to spin up a singular EC2 instance is relatively simple, in order to instantiate machines capable of functioning within an entire framework required additional efforts and trial and error before establishing a compatible slave. This files referenced below can all be seen in the Appendix also.

ConfigMaster File

The configuration master terraform file is arguably the most complex of the three scripts. One of the first issues encountered was the fact that the default Amazon Machine Image (AMI) used when beginning the investigation was an Amazon Linux machine, when it came to changing the AMI to an Ubuntu image as called out in the research design, the terraform scripts which had worked previously without issue were now failing to execute the `user-data` as expected. The `user-data` is a mechanism AWS offers to add custom bootstrapping commands to a machine as the final part of it's provisioning. This user-data's initial command of "`date >> provisionedAt.txt`" is intended to discern whether this phase of provisioning has run or not. This was working with amazon linux AMIs, but when changing the AMI image to Ubuntu and no other changes, the user-data would not run. This user-data is stored as a string and can be passed as such as a "`user-data`" parameter for an AWS resource or be base64 encoded and passed to the "`user_data_base64`" parameter either. Both of these were confirmed with the Amazon Linux AMI and both failed to work with the Ubuntu AMI. However, with the `cloud-init.log` on the node, which documents every step of the Amazon bootstrapping process, it was possible to see that the user-data was not running, as it was failing to find the shebang at the start of the script. (The shebang is the character sequence of `#!` at the start of a file that indicates it is an executable on a Unix based machine.) While this identified the problem with Ubuntu's parsing of the user-data, it was

inconsistent with what was happening on the Amazon Linux machine and also seemed to be a misleading error as the first line of the user-data already was "`#! /bin/bash`", a valid shebang. What caused the issue was eventually identified as white-space in the terraform script. The indentation that had been used for formatting and ease of review was causing the failure, and once removed, the portability between Amazon Linux and Ubuntu was addressed.

Another issue with the script came from the need to push a shared SSH key to the configuration master node to ensure it would have the ability to communicate with other nodes that it would be bootstrapping. While Identity Access Management (IAM) roles were originally looked at and are discussed in the section on AWS considerations, these would only be effective while the test harness was focused exclusively on the AWS platform. In order to enable the framework to target other environments, and without using usernames and passwords, a shared SSH key was needed. Avoiding the username and password combination was done due to the increased complexity in provisioning any new node to have a user with these details already established on the machine, and also as within a few hours of beginning investigations into how the tools would interact with various nodes, almost all defaulted to using SSH key authentication. To do this on Terraform required a specific type of "`provisioner`", a terraform concept which enables further modification to a machine after it's been created. In this case the framework is looking for a "`file`" type of `provisioner` which takes a locally accessible file from wherever the Terraform code is run and places it on the newly created instance. In this case, a local copy of the SSH key is identified by absolute path on the host machine and put in in a folder on the new configuration master.

But while this alone would theoretically enable the nodes to communicate with one another, to use the key in this fashion, it had to be possible to pass the key into any command that needed it. Considering the intention was to try and configure four different tools, each with their own nuances and library of commands, as well as any test framework specific functionality, this would become cumbersome and difficult to maintain. The solution was to extend the user-data script to validate if the ssh-agent on the machine was running. If not, it should start the agent and then to add the key to it as part of the `.bashrc` file for the default user. This `.bashrc` file is used to enable custom behaviour for the user of the home directory where this file exists and it is called when a new ssh session is created. This drastically reduced the complexity of using the key and would ensure authentication between node should not be compromised by the key file.

Node Files

The terraform files used to create the test nodes for this research were much simpler than the configuration master as the desired state was a default version of an operating system with as simple as possible a configuration. However there were some minor challenges to overcome when creating these files. The primary one of these was to do with the ability to execute a request to the registration API once the node was created. As one of the parameters required by the registration API was the IP address of the node once it had been created, this could only be done when the node was near the final stages of network provisioning. The initial attempts to get this running using the same user-data mechanism mentioned in the configuration master section failed in the compilation of the Terraform files when attempting to run them. This is due to the presence of the dynamic parameter `${self.private_ip}` in the request as seen by the full example of the command below.

```
"curl --retry 5 -X GET
 ↪  '${var.registrationAPI}?ipAddress=${self.private_ip}&cmTool=Salt&testSuite=4'"
```

While the `${var.registrationAPI}` code can be read in user-data as it's value is known when the script is compiled, the `self` family of parameters are only known after the code has been executed rather than compiled on AWS. This necessitated a new type of `provisioner` to be created, the `remote-exec` one in this case, which would be capable of running after the instance was created and would be capable of evaluating the private IP of the node that had been created during the execution phase. The remote-exec provisioner type runs after the user-data associated with the resource which also ensures that it will not impact the logging of timestamps as the node is created.

For Azure, this is more convoluted as it is a known limitation of Azure that the IP address for the machine is not known during the provisioning phase and instead it is allocated after this phase has finished. A method of bypassing this particular obstacle was found (Williamson, 2017) which utilised the Azure metadata API to obtain the private IP after creation. This was assigned to an environmental variable which was then referenced from `curl` command calling the registration API, combining variables declared in Terraform and variables created on the local machine. An example of this is below:

```
"export myIP="$(curl -H Metadata:true
 ↪  \"http://169.254.169.254/metadata/instance/network/interface/0/ipv4/ipAddress
    /0/publicIpAddress?api-version=2017-04-02&format=text\")",
"curl --retry 5 -m 120 -X GET
 ↪  \"${var.registrationAPI}?ipAddress=$myIP&cmTool=Puppet&testSuite=5&platform=azure\""
```

The other significant change to be added to the terraform files for the test nodes was the multiplier to enable multiple identical nodes to be created for testing purposes. This is luckily, something that terraform supports quite well with a single `count = 5` within the AWS resource block being all that was needed to add multiple nodes. The nodes within this count then used the `${count.index}` as part of the name of the server which was to be set as a tag. For Azure, this is slightly more difficult as the count variable must be assigned to each uniquely created resource for each machine being provisioned. The value for the count can be parameterised so it is a minor effort beyond the original set up to maintain.

4.1.2 Provisioning API

The registration API was one of the most elaborated components of this research and the implementation of this benefited from this low level analysis early on. The Lambda code for the API was written in Python 3.6 and for the majority of the code, it was simplistic, using Amazon's native boto3 library to allow the code to interact with AWS resources. Specifically in this case, to search through existing EC2 instances to find the target machines to work with based on the names used in the tags, allocated in the Terraform code. In addition to this, the `paramiko` library was used to grant the Lambda code the ability to connect via ssh to the identified nodes and submit commands. While this was straight forward to follow from the associated documentation, through trial and error, it was identified that only a single command could be executed through the paramiko connection via a Lambda call. Once this was identified, it was the driving factor for the creation of the bash scripts which will be discussed in the Bash file library in section 4.1.4

This reliance on bash scripts came with a price though, as the paramiko ssh session that is opened is acting like a standard terminal and the command will wait for some output before allowing the application to continue, even if the output is the simple return code of zero to indicate a successful run. This led to a number of issues that were ultimately solved once the root cause was identified through thorough investigation into AWS's CloudWatch Logging functionality and debugging commands in the AddTo Scripts. It was identified the Lambda code would simply return a timeout code when executing past the default time of 3 seconds.

Using CloudWatch, it is possible to output any desired information from the lambda code logged. As detailed in the Salt section, this Lambda function was calling scripts that were running a series of commands, including installing client components of the configuration management tools as well as the initial bootstrapping that accompanied them. In some cases, this could take up to half a minute to initialise a client node and until the Lambda timeout configuration was extended to accommodate that, the bootstrapping failed on a regular basis.

Arguably the most elusive root cause of all the issues within the test framework was the failure of the Lambda code to be accepted by the Lambda portal for the initial implementation. While the code worked locally, when the Lambda function tried to compile the uploaded code, it was regularly failing due to missing dependencies or incompatibilities among the shared libraries. After many failed debugging efforts, it took a single blog post by ElFakharany (2018) with similar symptoms to lead to the needed solution. The specific version of `pip` used to install Python packages is crucial in getting a compatible build with Lambda. Because Lambda runs on Amazon Linux at a low level, the solution was to spin up a dedicated Amazon Linux EC2 instance, copy the code and libraries to that machine to be compiled and once that was uploaded to Lambda, it worked without issue. With this compiled and working, the zip of the Lambda code was saved locally and any further updates to the Lambda code were achieved by modifying the zip's contents to just update the core python file, leaving the compiled dependencies intact.

Fig. 4.2 **Registration API behaviour by tool**

There were three other key considerations for the registration API once implementation began. The first was to add a third parameter to the API request other than the IP address of the node making the request and the configuration tool under test. This third parameter was a single digit designed to identify which test should be run as part of the provisioning of the nodes. The reason to separate the tests rather than running all the tests together was to give greater insights into exactly which elements of our testing would impact each tool the most as well as ensuring there was no interference among the tests themselves. The single digit

would be used to identify a test directory in the code repository cloned to the node upon start up in the user-data and tool-specific code was implemented to allow the tools to run the test within this folder.

The second change to the registration API was to allow the Lambda function to target the slave to initiate the tests as well as the master node depending on the technology being used as seen in Figure 4.2. This was done for tools where it was easier to run the tests, in particular if there were multiple nodes at once trying to obtain configuration to ensure that there was no conflicts among the calls to the master.

Finally, the third change was adding a final fourth parameter to the API request to identify the platform that had made the call to the API. This allowed the Lambda code to understand the different methods needed to identify the DNS name needed to proceed, namely using boto3 code to find the DNS name on AWS and using a static naming convention for Azure. This platform was also passed to the AddTo Family of scripts for dealing with variations there also, detailed in section 4.1.4.

4.1.3 AWS considerations

When the construction of the test framework was in it's initial phases, there were issues around permissions to access resources as part of the framework as well as reading and writing to shared components, like the S3 bucket used for logs. After some revision of the initial framework, it was clear that a series of IAM roles would assist in controlling what access was granted as well as avoiding the use of passwords or open resources capable of being accessed by the internet anonymously.

It is easy when working with cloud platforms like AWS to forget that it is another level of hardware and software outside of our control. AWS boasts of uptime of 99.99% in their own SLA, a rate so high that downtime is likely to never be seen by casual users or those engaged in short term projects like this research. However, on the 22nd of October, 2019, it was reported by AWS that their Route 53 service, a DNS resolution tool among others, was impacted by a DDOS attack (Spadafora, 2019). This resulted in many queries connecting to AWS's own resources and API entry points were inaccessible. This is what is used by Terraform to communicate with AWS as part of it's back end provisioning and because the errors are not capable of identifying that the service needed to communicate with Terraform themselves are down, a generic "could not connect" error was created. The end consequence of this was effectively a day spent debugging the test framework when in reality the issue was with the AWS platform itself.

4.1.4 Bash file library

While the terraform scripts will set up the config master and test nodes to a certain extent, in order 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 which keeps the the initial provisioning simple enough to prevent race conditions or timeouts. The scripts themselves also allow for a more parameter-focused process flow and prevent unnecessary duplication in the terraform files. There are three key files that we use in the framework outlined below.

setupBaseFiles.sh

This file is the only one called in the user-data in the terraform scripts and it's purpose is to ensure the configuration master is configured correctly for the tool being tested and any additional setup is done to enable the registration API to bootstrap the test machines. The tool under test is stored in the `${var.configTool}` variable in terraform and this is passed to this script as it's only parameter. Using this, the script checks what tool is being tested and installs any necessary services and modifies the associated configuration files also to allow an optimised set up of the tool under test. This file is also responsible for setting up the main log file on the master node, ensuring permissions are updated for any files the framework will use and then crucially, it copies the "addTo" family of scripts to a known location as well as updating their permissions for execution.

addTo scripts

The next set of scripts separated by tool. Each of the tools has a specific script named after it, so Ansible has "addToAnsible.sh" and Salt has "addToSalt.sh" etc. These scripts are executed when the registration API communicates with the targeted node and is one of the reasons we pass the configuration management tool to the Lambda function as a parameter. These scripts get two parameters passed to it when the Lambda runs, the test node IP address and the numerical test identifier. The script itself is tool specific and is the series of instructions needed to bridge communication between the test node and the master node. These steps are also logged individually to the central log file on the target machine when they are run. Once the setup in this script is completed, the final step within it is to call the next script we will discuss.

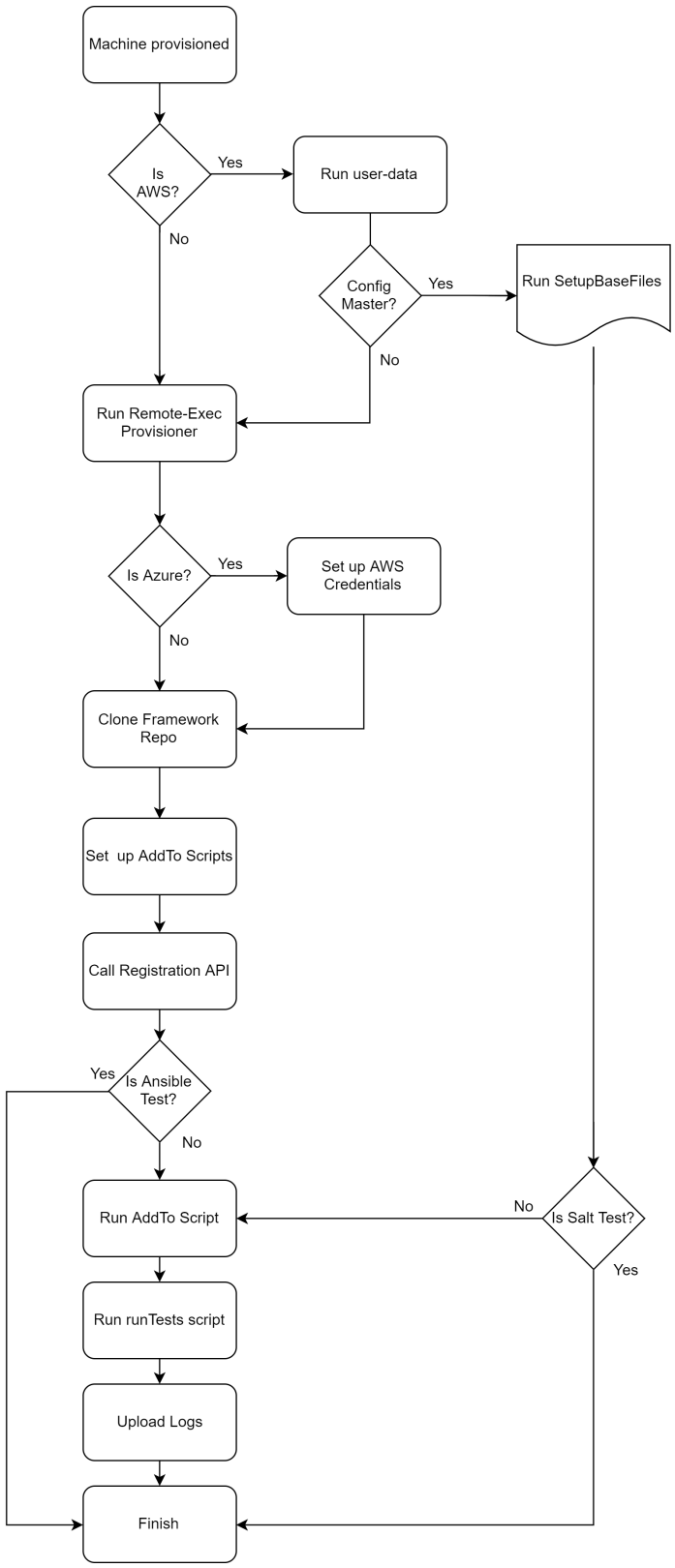

Fig. 4.3 **Process of provisioning and testing by node type**

runTests.sh

The runTests script accepts three parameters, the numerical test identifier, the tool under test and the IP address of the test node being targeted. Once executed, this script will determine which tool is being tested and using the test identifier, find the specific test scripts to be executed and then run the specific command to have the test node updated with the test script. At this point, because there are potentially multiple test nodes actively being tested, the central log file is at risk of becoming unwieldy and it would need some way to know when every node had completed testing before it was uploaded. A modification was made then to add the IP address of the node under test to a new log file which allows for more isolated log files which aid in reviews of the test results. This new naming format is important also because the final step in the runTests.sh file is to copy the log file containing the test results to an S3 bucket, using a directory structure controlled by the configuration tool parameter. This ensures that logs are safe and stored automatically and any test data available for specific tests can be aggregated for more tangible results.

4.2 Tools under test

The test framework is built around the necessity of running the same tests on a regular basis with as little intervention as possible and gathering the test results. However, a lot of the test framework is tool specific and it has evolved as the understanding of each tool increased while the research was ongoing. In order to be able to automate the set up of a master server, the bootstrapping of test nodes and tests themselves, each tool needed to be set up and manually have each step in this process researched, applied and documented. This exercise was the foundation for the automated scripts which enable each tool and it is worth noting how these impacted the test framework and the research overall.

4.2.1 Ansible

Ansible was the first configuration tool that the test framework was built to be compatible with and as a result, it was was effectively setting the baseline of how each tool would be compared. The set up of the Ansible master node was certainly the most simple, requiring only a repository to be added to the Apt package manager and then running `sudo apt install ansible` is enough to enable Ansible commands to be run. In order to connect to a test node, the IP address just needs to be added to the `\etc\ansible\hosts` file and then depending on the

`hosts:` specifier in the Ansible playbook, the file used to build our test cases for Ansible specifically, these hosts can be called individually or by logical grouping. This proved slightly problematic when the framework moved from supporting a single test instance to supporting multiple instances. All the documentation supporting the research on Ansible stated the need to add a header to identify a group of servers in the `\etc\ansible\hosts` file which could then be used as a parameter to pass into a the Ansible execution command. Because of this, the addToAnsible script checked for the existence of the expected header for the test in the file and if it didn't exist, it created it. If it did exist, it moved to check if the IP address it was attempting to add also existed in the `\etc\ansible\hosts` file. Again, if it existed, the script would exit and if it didn't it was added to the file. However, the command to execute the Ansible tests was originally calling the group of hosts when running, which was a known string to avoid confusion. An example of the command can be seen below:

```
sudo -u ubuntu ansible-playbook -i servers,
 ↪  /terraform/tests/Ansible/1-users/5users.yml >> /myLogs-171.12.11.85.txt 2>&1
```

The "`-i servers`" value passed to this command tells Ansible to check for all the nodes under the `[servers]` block in the `\etc\ansible\hosts` file and each of these machines is asynchronously updated with the commands in the `5users.yml` file. This caused issues when running with multiple hosts as each host would attempt to execute the above command and as a result, whichever node was bootstrapped first would potentially have the command run for that node as many times as there was nodes being tested. This necessitated a change in implementation where the IP address of the node executing the command would be fed into the command rather than the group identifier. This in turn led to an issue when the test files were executed as the playbooks require the `hosts:` specifier to match what value is passed to the command to indicate what nodes are being executed. When there is a mismatch, the value in the `hosts:` specifier actually informs Ansible on which hosts in the `\etc\ansible\hosts` file to ignore, rather than investigate, when looking to ensure the node referenced is one managed by Ansible itself. Because the `hosts:` specifier was a mandatory element of the playbook, the solution was ultimately discovered to be able to use `hosts: all` which effectively acted as a way to allow Ansible search through it's dedicated groups to confirm the node under test was one managed by Ansible.

The only major issue that occurred with implementing Ansible as a tool was an issue with dependencies. Ansible itself runs primarily on Python, although it can function without it if needed, but this is an exceptional circumstance and used almost exclusively to install Python on a target node. While the Ubunutu instances selected for this test came with Python,

their default version was Python 3.5. Ansible supports both Python 2 and 3 but the version installed when the setupBaseFiles.sh script is run is 2.9.1. This version targets Python 2.7 by default and has many of the commands are configured only to work with this version of Python. In order to allow this version of Ansible communicate and update the test nodes, they would have to have their default version of Python changed from Python 3 to Python 2. Some quick experimentation found that it was faster to run an install of Python, which makes the latest version installed the default version to be faster than using something like virtualenv to modify the default versions. As a result, for the test nodes when running Ansible tests, part of the user-data files include a command to install the 2.7 version of Python on the test node also, ensuring full compatibility with the version of Ansible installed on the master node.

As part of the setup for these tests, there were a number of updates to the `ansible.cfg` file, the main configuration file for Ansible. This ensured that the testing proceeded without the need for manual intervention in places as well as adding a higher degree of visibility into the timing of each activity in the output log generated by Ansible. By setting `callback_whitelist = profile_tasks` in this file, each task that ran would output how long in milliseconds the task took. It would also put these this information together with a summary at the end of the log which simplified the analysis of the tests. Of all the tools that were set up, Ansible was the only one which did not output these timings by default, but the simplistic configuration updates made this additional information possible.

4.2.2 Chef

Of the tools that were investigated for this project, Chef was by far the most complex to set up and get running. Initial research into the tool outlined the various options that the provider offered, each with it's own dedicated set of configurations again to cater to any size infrastructure. For the relatively small number of nodes used as part of this test, it would likely make sense in a commercial setting to use chef-solo, a concept that would see each node install a Chef client and pull the necessary test files, in Chef's case called recipes, to the node and apply them via the local client. This master-less version however is limited in the functionality Chef offers and is unlikely to be used in enterprise infrastructure where a centralised hub to control all of the corporate technology is much more efficient. Since this research is targeted at how to ensure the most cost efficient tool is recognised, and this is most relevant to commercial entities, this version of Chef was investigated just to identify the variances between it and the standard infrastructure automation tool Chef offers.

Once the specific entity of Chef was identified to be used with this research, an investigation into how to set up a simple Chef master and test node began. This proved to be incredibly problematic for a number of reasons. Initially there was an attempt to leverage the Amazon provided configuration management service known as OpsWorks (Zhu *et al.*, 2014). This is a service that provides a dedicated AMI to bootstrap configuration masters for either Chef or Puppet and allows an associated auto-scaling groups in Amazon's EC2 infrastructure to be added automatically or it can be run and nodes can be added after the service is ready. On paper, this seems like a ready-made solution, perfect for this use case we're experimenting with and already highly integrated with the cloud platform used by the test framework. However with this server set up, and so much of the configuration hidden upon start up, changing anything once the Chef server was ready was difficult and not a huge amount of information was made available to help understand this.

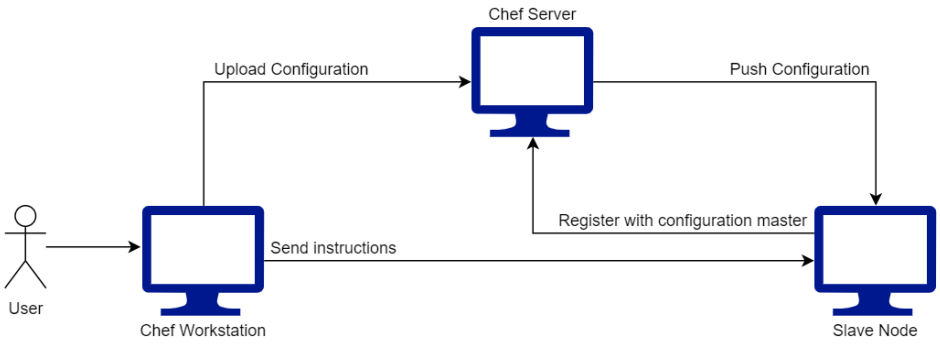

Fig. 4.4 **Set up of Chef configuration management architecture**

Conflicting information was a common occurrence while doing research for Chef and many guides and blogs reviewed in order to get the server set up contained references to functions no longer available in the version of Chef being installed or mentioned changes to configuration files which also no longer existed and were not used by Chef anymore. In several cases, while the instructions to set up Chef's master server worked, the documentation around generating tests or connecting the slave nodes would be obsolete or require libraries or dependencies that have since disappeared from repositories and file servers since the instructions were written. Eventually, following the instructions from Chef's learning portal with some collaboration with the Chef community Chef (2019a), a Chef master server was created, coupled with a "Chef workstation", a concept that separated the trusted repository of all the recipes to be shared among the infrastructure once committed on the Chef automate server and the work in creating the recipes and initiating the push of configuration out to slaves via the workstation. A representation of this can be seen in Figure 4.4. The Chef learning portal is

highly recommended to ensure the most up to date version of the documents are referenced, as many of the third party information is now out of date.

Because the configuration for Chef is reliant on Fully Qualified Domain Names (FQDN) to identify the Chef server and each associated connection, including that of the workstation, this presented some difficulties in automation. The set up of the server and workstation were suitably lengthy and complex that for the simple act of stopping and restarting the nodes alone would generate new dynamic IP addresses in AWS, which would in turn modify the FQDN for both machines and break the configuration. As a result, due to time constraints, the servers for Chef were the only ones left running for the duration of this research as the effort needed to automate this was not within the scope of the project and couldn't be achieved within the project schedule.

Once constructing the actual recipe files, Chef's default file type to enforce a particular state on a machine, it was evident from the amount of documentation supporting them that the ruby-based syntax used for these tests offered a huge amount of flexibility in their creation. The structure was intuitive and the majority of the investigation into the recipe creation went towards identifying keywords and understanding the capabilities of the language. Chef offers an example pack when the tools are initially installed as part of the set up and this example pack was used for initial configuration and testing. For reasons still unknown, the commands offered by Chef's documentation to generate new cookbooks, the folders where recipes would normally reside, were not working as expected and failed to generate any resources. However one of Chef's primary marketing strategies towards new users is it's "Chef Supermarket", a library of commonly used Chef recipes available for use by anyone for free (Chef, 2019b). These are accessible through Chef's command line tools or can be downloaded as a zip file and for the purposes of this research, recipes fulfilling the test criteria or similar enough to require minor modification were available for use.

4.2.3 Puppet

The research for Puppet to establish the necessary understanding of the tool as well as execution of the tests was relatively straight forward and accomplished with some useful documentation provided by the Puppet community such as Kernal (2019) and Hong (2018). Disregarding some initial confusion over Puppet's heavily focused push on enterprise solutions, an investigation into the open-source version of puppet revealed useful documentation for set up and configuration. As this was done after Ansible and Chef, there were some parallels to draw upon of these, for example the tendency for all of the Puppet documentation

to encourage use of FQDN names is reminiscent of Chef and the use of the hosts files is similar to how Ansible tracks its nodes. This combined with Puppet's availability as a package in the Apt repositories allowed for a set up that had a successful ping between master and slave within a few hours of research, in comparison to several weeks of work before Chef managed to achieve two way communication.

Where Puppet became more difficult to work with was it's module files; the files that would contain the commands necessary to complete the tests. Puppet itself is simplistic to use and with very basic functions, the modules do not require much in the way of configuration. However, in more complex scenarios, the recommendation from Puppet themselves is to leverage the Puppet Forge, a utility similar in nature to that of the Chef Supermarket. While Supermarket offered a fully encapsulated package with recipes ready to run, Forge is more likely to provide users with a class like interface that Puppet uses to extend it's base functionality. These operate in such a way to offer new keywords and attributes that are not natively offered and abstract a lot of the complexity needed to accomplish the desired effect in usually no more than 3 or 4 lines of puppet syntax as seen below.

```
class { 'java':
  jdk     => true, # default - whether to install the jdk or the jre only
  version => '8', # Java version to install
}
```

4.2.4 Salt

It may be inferred that at the point of investigating the fourth of the configuration tools discussed in this research that domain knowledge may have increased sufficiently to enable adoption of each tool at a more rapid pace throughout this project. However, it would be remiss to understate the simplicity Salt has achieved in it's installation process. From the beginning of the investigation into Salt, it was quickly clear that usability was something kept as a high priority for administrators of the configuration management tools. Within half an hour of starting the Salt research, communication between a Salt master and Salt minion was achieved. The offering of a publicly available bootstrapping file from the company behind Salt themselves, which simply required the DNS address of the Salt master to successfully register a node, made this work quick and easy to achieve.

While some of the tools require very little in set up for the test framework, in other cases, some workarounds had to also be implemented to facilitate automation as part of the testing process.

Salt is an example of this in that, although it had a quick manual set up, the framework needed to be modified in a number of ways to allow automated testing. The first was an authentication issue from the bootstrapping functionality. Although Salt documentation stated that setting `auto_accept: True` in the `/etc/salt/master` file would prevent the need for having manual approval of each node upon bootstrapping, this did not work as expected. During the bootstrapping process, a minion registers it's intent to connect to a master node. This can be seen on the master node by running the command `sudo salt-key -L` where the keys that have requested admission are seen as being unaccepted as evidenced by Figure 4.2 below

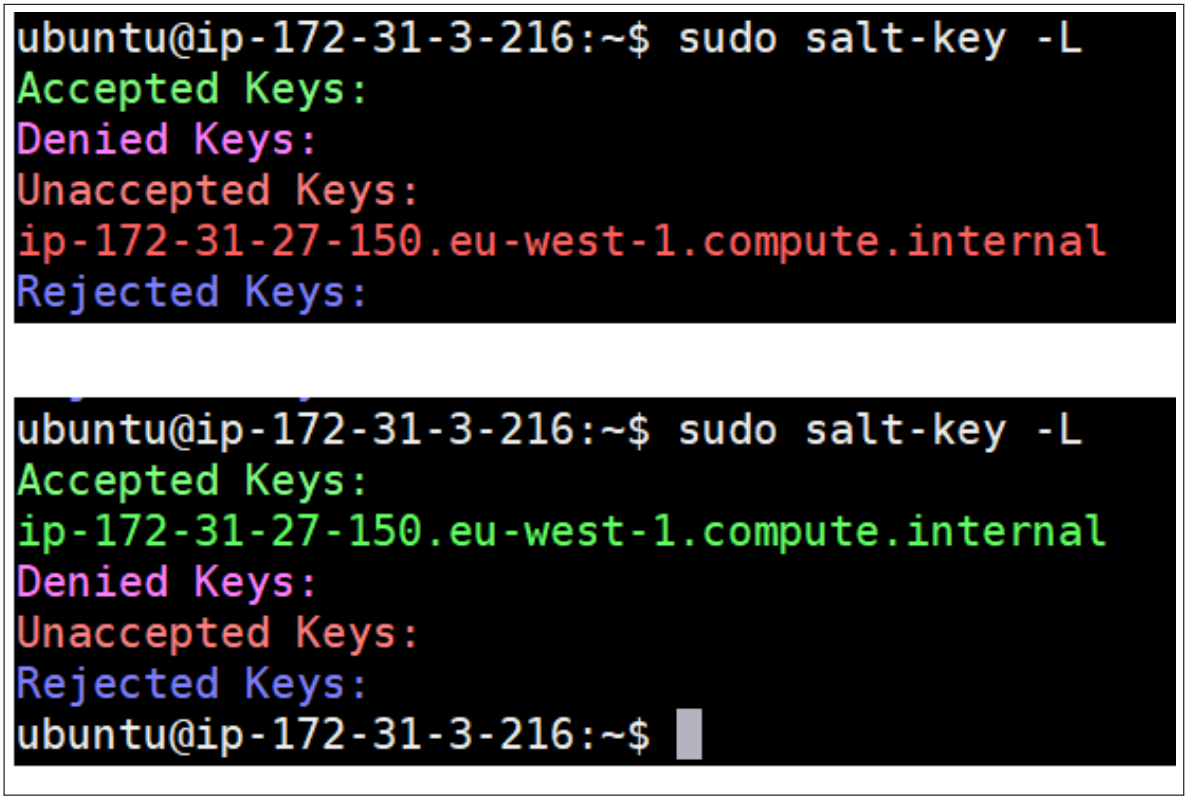

Fig. 4.5 **Salt Master showing status of Node before and after key was accepted**

In order to get the key to be accepted, as the `auto_accept: True` command did not seem to be working, it was then necessary to embed a simple script that would run "`sudo salt-key -y -A`" every second on the Salt Master itself. This would ensure that the Salt Master would not leave any node in a state where it was not accepted once it's authentication request was accepted. This script was written in very simply bash shell scripting and it's footprint on the Salt Master was negligible in terms of the Salt Masters ability to execute any necessary commands to run the tests and would not impact performance in any meaningful way.

Salt also changed the way the framework worked from a registration API perspective. Because the minions needed a level of bootstrapping in order to connect to the master node, the registration API which was configured to connect to the master to add a minion node, was modified for Salt use cases, as mentioned in the section on the registration API. In the scenario where Salt was the tool under test, the registration API would instead re-connect to the minion and begin the boot strapping process. While this could potentially have been done by modifying the setupBaseFiles.sh script also, as the change was relatively small for the Lambda code, this was the option chosen. This did lead to some debugging though as the addToSalt.sh script was now executing the bootstrapping code which prevented the return code returning to the Lambda until the bootstrapping had completed and therefore allowing the addToSalt.sh script to finish with a return code, something the paramiko library required to close the connection, and the cascading effect led the Lambda function which had a default timeout of 3 seconds to fail regularly until this was identified as the cause and the timeout was subsequently modified.

The other reason that Salt had it's minions targeted rather than it's master by the registration API was that it was simpler to execute the tests. Each of the tests for Salt was contained in a single state file, Salt's equivalent of playbooks, recipes or modules. In order to access these state files, the Salt master needed to acknowledge a "file root", a directory location that would hold these state files for the tests, as well as a parent `top.sls` file which referenced each state file in turn. This defaults to `/srv/salt/` for standard configurations of Salt. With these identified, in order to execute a test from the Salt master would require getting the minion's identifier, in this case the internal DNS name for each minion provided by AWS. While possible, to execute the test from the minion simply required the following command:

```
sudo salt-call state.sls jdk
```

This would go to the salt master, search the file root directory where the `top.sls` file resides and search for a file called `jdk.sls`. Due to the ease of this approach, this was the one adopted by the test framework.

4.3 Test Case Considerations

Throughout the process of setting up each of the tools, as the tests were being implemented, it was necessary to start to elaborate exactly what the actions were that each test would consist

of. Some of these decisions were based on trying to determine valid yet achievable scenarios that may exist in commercial entities. Others were further modified after seeing that the configuration tools were completing the actions so quickly as not to get useful information from the tests. Here we will outline quickly what will actually be implemented in each test.

1. Users - This functionality will create a linux user on the slave under test. This test was one of the quickest to execute and as a result, it was determined to try and ramp up the efforts needed to accomplish this test. Rather than creating a single user, this test now creates 10 users, each with an associated home directory and unique UID.

2. JDK - Rather than trying to install Python and Java, this test was reduced to simply using the Apt repo to install openjdk-8 on the node. This alone was seen as intensive enough as it's own test and having a single action in a test would make it easier to try and discern where any differences in time may be stemming from. As the JDK itself is only approximately 1.6MB in size, the configuration and installation steps of the apt package installer are what are more likely to contribute to testing times.

3. File Management - Undoubtedly the most efficient test on each tool when initially tested. As a result, this has bumped from a single file to 5 folders with 10 empty files in each. This places slightly more pressure on the tools but this is still not significant. A script to generate as many folders and files as desired was created to accompany the test framework code to potentially create thousands of files if stress testing was required in this area.

4. Repositories - Here the test is targeting the time taken by the slave to use git to download a repo into an initialised folder. Originally this test was only going to clone a single repository to the node under test but due to incredibly quick performance when cloning the single repository that the research code for this project was in, it was determined to that this test needed an additional repository to work with. This test now also clones the kubernetes repository held in github, seen here -> https://github.com/kubernetes/kubernetes

5. Webserver - Conveniently offered regularly as an example test for most of these tools, setting up an Apache web server, a scenario very likely for many use cases, this test will only download, install and validate that the apache2 service is running, it will not attempt to set up a webpage or validate that hosting a webpage works.

6. Restart - This test was ultimately removed from the test suite as no tool was capable of reporting back the restart process as the nodes require an unbroken connection to the

master at all times to report back a status. Restarting a node will break this connection and therefore this test was no longer used as part of this research.

4.4 Summary

The implementation steps and knowledge outlined in this chapter build on top of that of the previous research design chapter in order to give a more technical overview of the work done as part of this research. This information was gathered as the research itself was completed and should inform at a lower level at how each of the components work and interact with one another. In the next chapter, the results of this research implementation will be discussed.

Chapter 5

Results

Chapter 4 was primarily concerned with the methods and approaches utilised in order to accomplish this research. The low-level implementation detail allows for a more thorough understanding of the various levels of interaction between the components and how the tests themselves were executed. In this chapter, we will examine the results of this research, the metrics gathered, and explore the implications of the findings within the parameters of this study.

5.1 Timings

As per the plan called out in Chapter 3, the framework will attempt to log timings of each of action with the intention of giving a realistic timeline of the life cycle of the experiment. This is done with the simplistic use of outputting the `date` command followed by a relevant description of the timing being logged to a series of log files. This lightweight approach was chosen due to the lack of potential impact on the test machines, as no additional packages or processes would be needed in order to register the checkpoint each time it is called. By default this command will output a timestamp down to the second which is deemed low level enough for this aspect of the study. As we move to look at the timings from the tools, we will see many of the tests execute quickly enough to warrant a more detailed unit of measurement, but for the sake of normalisation of the data, all timings will be represented as seconds in decimal form.

Some of the key metrics captured are outlined below in Table 5.1 and labelled for ease of review. The framework does log more timestamps than those outlined in the table and can be

Table 5.1 Triggers for timestamp creation

Timestamp name	Timestamp Trigger	Tool or Framework
Provisioned At	Node created and bootstrapping started	Framework
Running Test	Bootstrapping complete and test starting	Framework
Time Taken	Test execution time	Tool
Finished At	Confirmation test is complete	Framework

seen in the raw log files, but these are primarily for debugging and diagnostic purposes. The derived knowledge from this research, the key metrics to be reviewed and discussed are all gleaned from the timestamps listed below.

5.1.1 Logs

For each test run as part of this research, there is a part of the framework that will ensure that the last step of the logging process is to upload the file to an S3 bucket, identified within the framework itself. The file is copied up with the knowledge of the IP address of the node under test as well as the tool performing the test. This is used to ensure the file is uploaded to a directory that allows for easy review, using the tool name as a directory, and then allowing for any significant anomalies in the test results to have an identifier for the node so investigation and debugging can occur as needed. For Ansible and Salt, retrieving the logs only necessitated the upload of a single file as the core timings are all gathered from the master and slave node respectively. Puppet though, as seen in Chapter 4, necessitated setup activities on both the master and slave node and as such, two files were uploaded for Puppet, one from each of these machines.

Figure 5.1 is an example of the test log file generated by the framework and what information it tracks. Key times that are tracked are those for `*** Running Test ***`, `*** Finished Test ***` and the total time taken by the tool itself to execute the test in question. As part of the validation of this research, the time difference from a test completing and beginning was compared to the test tool's own execution time. This was intended to ensure that none of the tools were reporting times which were inaccurately making them look more performant. Through this collection of research, this was confirmed to not be the case for any of the tools under test, making their timings more authentic for our validation.

```
Mon Oct 14 02:08:33 UTC 2019 Log File created and permissions set
Mon Oct 14 02:08:33 UTC 2019 AddToFamily scripts copied and permissions set
Mon Oct 14 02:09:48 UTC 2019 not found and added 172.31.29.136
Mon Oct 14 02:09:48 UTC 2019 running sudo bash /terraform/scripts/runTests.sh 1 Ansible
Mon Oct 14 02:09:48 UTC 2019 ***Running Test***
Mon Oct 14 02:09:48 UTC 2019 Executing: /terraform/tests/Ansible/1-users/5users.yml

PLAY [servers] *****************************************************************

TASK [Gathering Facts] *********************************************************
Monday 14 October 2019  02:09:49 +0000 (0:00:00.068)       0:00:00.068 ********
ok: [172.31.29.136]

TASK [Create the test group] ***************************************************
Monday 14 October 2019  02:09:51 +0000 (0:00:01.676)       0:00:01.744 ********
changed: [172.31.29.136]

TASK [Create test User1] *******************************************************
Monday 14 October 2019  02:09:52 +0000 (0:00:00.980)       0:00:02.724 ********
changed: [172.31.29.136]

TASK [Create test User2] *******************************************************
Monday 14 October 2019  02:09:53 +0000 (0:00:00.920)       0:00:03.645 ********
changed: [172.31.29.136]

TASK [Create test User3] *******************************************************
Monday 14 October 2019  02:09:53 +0000 (0:00:00.409)       0:00:04.054 ********
changed: [172.31.29.136]

TASK [Create test User4] *******************************************************
Monday 14 October 2019  02:09:53 +0000 (0:00:00.349)       0:00:04.404 ********
changed: [172.31.29.136]

TASK [Create test User5] *******************************************************
Monday 14 October 2019  02:09:54 +0000 (0:00:00.346)       0:00:04.750 ********
changed: [172.31.29.136]

PLAY RECAP *********************************************************************
172.31.29.136              : ok=7    changed=6    unreachable=0    failed=0

Monday 14 October 2019  02:09:54 +0000 (0:00:00.326)       0:00:05.077 ********
===============================================================================
Gathering Facts --------------------------------------------------------- 1.68s
Create the test group --------------------------------------------------- 0.98s
Create test User1 ------------------------------------------------------- 0.92s
Create test User2 ------------------------------------------------------- 0.41s
Create test User3 ------------------------------------------------------- 0.35s
Create test User4 ------------------------------------------------------- 0.35s
Create test User5 ------------------------------------------------------- 0.33s
Mon Oct 14 02:09:54 UTC 2019 ***Finished Test***
Mon Oct 14 02:09:54 UTC 2019 ***Uploading Test Results***
Mon Oct 14 02:09:54 UTC 2019 Executing: aws s3 mv /myLogs.txt s3://dhill-config-management-tests/debug/Ansible/myLogs-Mon Oct 14 02:09:54 UTC 2019.txt >> awsCopy.log 2>&1
```

Fig. 5.1 **Sample Log File**

5.2 Test Results

In the following sections, the results of the tests will be shared and discussed. 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 staggered timeline with irregular numbers of nodes under test and across the two cloud platforms of AWS and Azure are only valid for Ansible, Puppet and Salt. In each test, the timings collected under "Manual Execution" are derived from the first successful execution of a test, done to understand the steps to automate, seen in section 4.2. These timings are included for full disclosure to see if there are variations that may have been introduced by the framework. It is also critically important to remember these timings are all gathered after the provisioning of a node has completed, so the code used by Terraform has no bearing on the results and the files executing the tests are timestamped to ensure only valid timings are captured.

Due to time constraints and the complexity of Chef, it was not feasible to integrate it into the test framework. However, as part of the review of each of these test results, the timings gathered during the initial manual implementation of these tests will be examined and these will include timings for Chef. While these are not as thoroughly investigated, a level of insight is available from their presence and this data will feature in the end results, albeit with less confidence in the metrics presented.

5.2.1 User Creation

In Table 5.2, the first of the results for the tests defined in Chapter 4 can be seen. For this test, 10 unique users are created with their own UID and home directories, the number of users increased to 10 as it was deemed a single user is not just an unlikely business case for larger infrastructures, but also making it easier to ascertain timings for the tools after a review of the timings collected when a single user was the intention of the tests.

Very quickly, it is evident that there is a significant difference in the performance of the tools, With Ansible being over 15 times slower than Puppet, the fastest of the tools in this test. However, the first example of the variance which impacts the confidence in getting iron-clad metrics is already visible when comparing the difference in timings from the initial manual execution and the final results collected as an average of multiple test runs and many nodes. Ansible here has the biggest change, with 5.7 seconds as the timing recorded in the initial manual run. But during the automated tests used to collect data for this report, the

fastest time Ansible was able to complete the same test was in 8.22 seconds. This variance is still unaccounted for why the manual run was so performant initially but the timing was ultimately unable to be reproduced in the framework.

Puppet on the other hand was quite consistent throughout its runs for this test with a range only spanning 0.21 seconds from it's fastest test time (0.57 seconds) and slowest test time (0.87). Salt was another example where the initial test time was completely different from the aggregated test results, with the manual execution being 1.4 seconds faster than anything achieved during the test framework's execution. However, from a research perspective, the timings offered by Salt and Puppet in this test are fast enough that neither would warrant further investigation. The Ansible timings however in this case, with an average time several multiples of any other tool on offer is an initial insight into the trade-off for ease of use over performance.

Table 5.2 Time in seconds to create 10 standard Users

Tool	Manual Execution	Averaged	Fastest time	Slowest Time	Variance
Ansible	5.70	11.04	8.22	12.95	1.78
Chef	3.00	N/A	N/A	N/A	N/A
Puppet	0.58	0.66	0.57	0.87	0.007
Salt	0.47	1.95	1.86	2.20	0.009

5.2.2 JDK Installation

Table 5.3 outlines the results for the JDK installation test. This test was primarily focused on how each of the tools would report back on the `apt` commands which were run in the background behind the syntax that each tool followed. As the test itself has two simple requirements, update the Apt repositories and then use `apt` to install the same package in each test, the expectation was that the timings would be similar as the majority of the time taken would be from the installation steps themselves and that any discrepancies in the timings would be a result of how each tool deals with the submission of the commands as well as interpreting the return codes and how it validates that the activity has worked. Investigating how the validation is achieved is not within scope of this research but it is an assumed that this validation can be trusted and that each tool will only return status and timings once this has completed.

The focus of these results is on the jump for Salt from it's initial manual run of 27 seconds to having an average timing of 189 seconds in the automated framework, 7 times the earlier

runtime. While the other tools have remained relatively constant in their timings with only slight variance, the average figure of Salt comes from a range in the framework which starts at 35 seconds and goes up to 293, a difference of almost 4 minutes. This variance when compared to Ansible (29 seconds to 34 second) and Puppet (24 seconds to 29 seconds), shows Salt as a tool which not only performs at a slightly slower rate in this test, but also with a significant variance that would make it very hard to predict how long any activities would take.

Table 5.3 Time in seconds to install openjdk-8

Tool	Manual Execution	Averaged	Fastest time	Slowest Time	Variance
Ansible	36	32	29	35	2.93
Chef	61	N/A	N/A	N/A	N/A
Puppet	25	26	24	28	2.09
Salt	27	189	34	293	8194

Since we know that all of these tools take at least 24 seconds to complete at minimum, we can infer that for apt to install the JDK will take less than this, as any tool which submits this command must translate from it's proprietary syntax to native commands, and the validation by the configuration management tool will also contribute to the overall timing. This dictates that the lengthy time taken by Salt must not be because of the installation command but supplementary activity that Salt hides from the end user. One possible explanation for this is that both Puppet and Ansible specify the `apt` command explicitly for use as a package manager. Salt however is agnostic with it's syntax as seen below.

```
uptodate:
    pkg.uptodate:
        - refresh: True

openjdk-8-jdk:
    pkg.latest
```

An investigation into Salt's open source code repository show's checks by the tool to examine the underlying operating system that a minion is installed on to best understand what commands should be used to achieve it's objectives when interpreting commands from a Salt Master. The module used by Puppet has a specific java installation module, shared through the Puppet forge, Puppet's online shared repository for scripts to achieve popular objectives. It has very specific checks that are similar to those seen in Salt's source code, but Ansible is

coded to use the Apt package manager explicitly. This means that Ansible's playbook for this test is not as portable as the custom built Puppet module or Salt's generic validation, built into the tool, and the cost of this portability being built into Salt is evidently that of performance.

5.2.3 File and Folder Creation

Of the tests completed as part of this research, this test to create five folders and ten empty files within each folder is the one that highlights the difference between these tools when it comes to native operating system activities. Creating files and folders is an extremely common task and it is common knowledge that the execution of the commands to do these actions are almost instantaneous.

Table 5.4 Time in seconds to create 5 folders and 50 files

Tool	Manual Execution	Averaged	Fastest time	Slowest Time	Variance
Ansible	15.6	44.0	27.5	48.0	35.75
Chef	1.00	N/A	N/A	N/A	N/A
Puppet	0.90	0.11	0.10	0.12	0.000009
Salt	0.32	0.35	0.29	0.43	0.0015

While most of the tools manage to complete this activity in times that would indicate almost native levels of execution, Ansible's execution times, seen in Table 5.4 are significantly higher than any other tool, in the automated testing of the framework getting as high as 48 seconds. When compared to the times shown by Puppet and Salt, both of which manage to get the task done in comfortably under half a second, this highlights that Ansible's method of execution suffers from slow performance. We know Ansible is built on Python, so it is possible that there is a delay due to translation but this is arguably the test case that show's Ansible as the most unlikely candidate for any considering a tool to manager their infrastructure.

5.2.4 Cloning two Git Repositories

The purpose of this test was to use a third party tool, in this case `git`, through each tool under test to understand how it may affect the execution times. Git itself was installed on each node by default as it is packaged with the version of Ubuntu under test, meaning that the only commands needed for the test to be deemed successful is the `git clone` command. This test will clone the contents of the repository holding the code for this research as well as

the publicly available Kubernetes code base to ensure there is enough content to generate meaningful results.

Table 5.5 Time in seconds to clone two repositories

Tool	Manual Execution	Averaged	Fastest time	Slowest Time	Variance
Ansible	96	96	90	100	11.22
Chef	125	N/A	N/A	N/A	N/A
Puppet	133	99	87	123	147.77
Salt	146	624	92	837	79686

As seen in Table 5.5, these results are relatively consistent across the tools in the initial runs and for Puppet and Ansible, this doesn't change when the test framework automated this test. Salt however, has massive inconsistencies, capable of running in under 100 seconds for some runs and then taking over 800 seconds for other runs, similar to the behaviour seen in the JDK installation test. It is worth noting at this time that for this test in particular though, Salt had a consistently high test execution time, of over 14 minutes per run for set period while doing these tests. This period of execution took place over 24 hours and ran on 10 different instances during this period. Since then, these extremely high timings have not been reproducible but as they cannot be discounted due to any known issues within the framework or tools, the timings have been kept within the collected metrics and aggregated within the average time. Without these runs included, the average timings fro this test for Salt is 110 seconds.

5.2.5 Apache HTTP Server Setup

The final test performed within this research was based around the activity of not just installing package on the node in question, but also ensuring that it was activated for potential business use cases. Arguably one of the most traditional setups for remotely managed infrastructure is that of servers running Apache HTTP Server to host web content.

In contrast to some of the more spectacular differences in execution time seen in the earlier tests, this test shows relative consistency across Puppet and Ansible with very little variance between the initial manual runs and the final averaged values. Salt does manage to get an average time which triples the initial run but when the timings that make up that average are examined, there are a number of runs that complete in just over 10 seconds and some of the runs took between 65 and 70 seconds, with a number of execution runs taking times between both ends of this range. This variance that Salt suffers from is not present to the

Table 5.6 Time in seconds to set up Apache HTTP Server

Tool	Manual Execution	Averaged	Fastest time	Slowest Time	Variance
Ansible	21.0	26.4	20.1	37.6	41.58
Chef	20.0	N/A	N/A	N/A	N/A
Puppet	7.37	6.72	6.17	8.00	0.2899
Salt	10.2	36.8	10.0	69.8	553

same extent with Puppet and Ansible, both of which manage to keep their variance levels low and therefore keep their timings consistent.

5.3 Azure

The testing done on Azure was done with the intention of ensuring that there were no platform specific biases towards any of the tools under test and to see if there was any interesting variance between AWS and Azure execution times. While the code to create the virtual machines in Azure is significantly more detailed than that of AWS's EC2 instances, the specification on each type of machine was as similar as possible, using the same version of Ubuntu, installing the same versions of the tools and running the same tests. The cloud platforms offer region variables to try and minimise network latency, and for both of these platforms, there is an option to use the data centres based in Ireland, and this was selected throughout the research.

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 5.7, but there were some anomalies also. In particular, the installation of the JDK seemed to take significantly longer on Azure and this was one of the tests run more than once 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 to initialise the webserver on Azure, but with the variance of Salt in general, this is not a solid indication that it would always be faster on Azure.

Table 5.7 Comparisons of average times on AWS vs Azure

Tool	Test	Average on AWS	Average on Azure	Variance including Azure
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

It is worth reinforcing that the differences between AWS and Azure could possibly be influenced by the fact that the Master server lived in AWS throughout this research and as such, there is likely to be less latency between AWS machines than those spanning multiple providers. However, the fact that the timings are intended to only track the activities of bootstrapping the slave node, the only expected additional latency would come from the initial communication to the master node and potentially reporting back the status. For some of the test results seen in Table 5.7, where the difference is only one or two seconds, this seems reasonable. But this would not explain the much larger variations and this itself warrants further investigation.

5.4 Summary

While there is likely no definitive and concrete metrics which exist as part of this research, what this study can do is identify trends among the data gathered and interpret likely attributes and data-driven decisions to inform consumers of this research. From the metrics gathered by this effort, it appears as though Puppet will consistently perform well in comparison with it's peers, only being out paced on rare occasions but never significantly, usually only by a few seconds. 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 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, significantly so. A tool that is capable of taking ten times longer to do the same task based on unknown variables is a cause for concern and may have impacts on things such as disaster recovery and scaling at speed, concepts that configuration management is often marketed as the solution for.

The results of this research are merely a snapshot of what a specific version of each tool is capable of, in order to give a level of insight into what the current trends are and how they might inform potential users of what impact the choice of tool has on potential bootstrapping times for their infrastructure. While efforts have been made to keep these tests uniform by eliminating variables where possible (such as using the same version of Ubuntu on each node, using the same test specification across each tool and the same success criteria for each also), these measures are not capable of ensuring absolutely identical variable within each environment and test. By it's very nature, cloud computing is dynamic and unlike physically managed servers, it is not possible to know what competing resources may be shared by any node within this research. As seen when looking at the tests running in Azure, the change in platform does have an impact in execution times and this should be taken into account.

This potential difference in environmental variables and other unknown contributing factors have been mitigated by repeated testing, a capability delivered by the test framework delivered as part of this research. As such, that framework, in conjunction with the results obtained through it's use, is as much an artifact of this research and will be made available for future work. This foundation laid by the framework has allowed for significantly more testing to be possible than without it but it would be implausible to assume that the time allocated would be sufficient to address every possible factor in testing and as seen in the future work in section 6.3, there is plenty of growth for this framework to isolate the test timings as much as possible. What this framework has achieved though is a capability of executing these tests with as much automation as possible, needing only two terraform files to be executed, for a total of 4 commands, which can be further automated easily enough to run on a automatic timer in future.

For further review of the logs reviewed as part of this study, the logs can be found at the location below.

Log Location

https://s3.amazonaws.com/dhill-config-management-tests/testResults

Chapter 6

Conclusions

This study was based on the concept that optimal configuration management is something that can have an operational and potential fiscal impact on a commercial entity and, therefore, there was merit in investigating how some of the most popular configuration management tools performed in comparison to each other through a controlled set of experiments. In order to tackle this research in as objective a fashion as possible, it was determined that consistent, repeatable and regular testing would be necessary and as a result, the output of this study is a combination of the framework designed to facilitate this study as well as the performance metrics gathered during the testing. Throughout the research, data on the usability, complexity and ease of integration for each tool was also documented, giving further insight into how the ultimate decision of "which configuration management tool is suitable for a specific enterprise" may be influenced.

6.1 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 of a said task. It also is the arguably the most consistent tool, having the least variance among each run it performed, it would be possible to narrow Puppet's expected execution timing range to a much narrower spectrum. This will help maintainers of the infrastructure debug issues and gauge the progress of tasks, granting a higher level of confidence in the tool's ability to manage change.

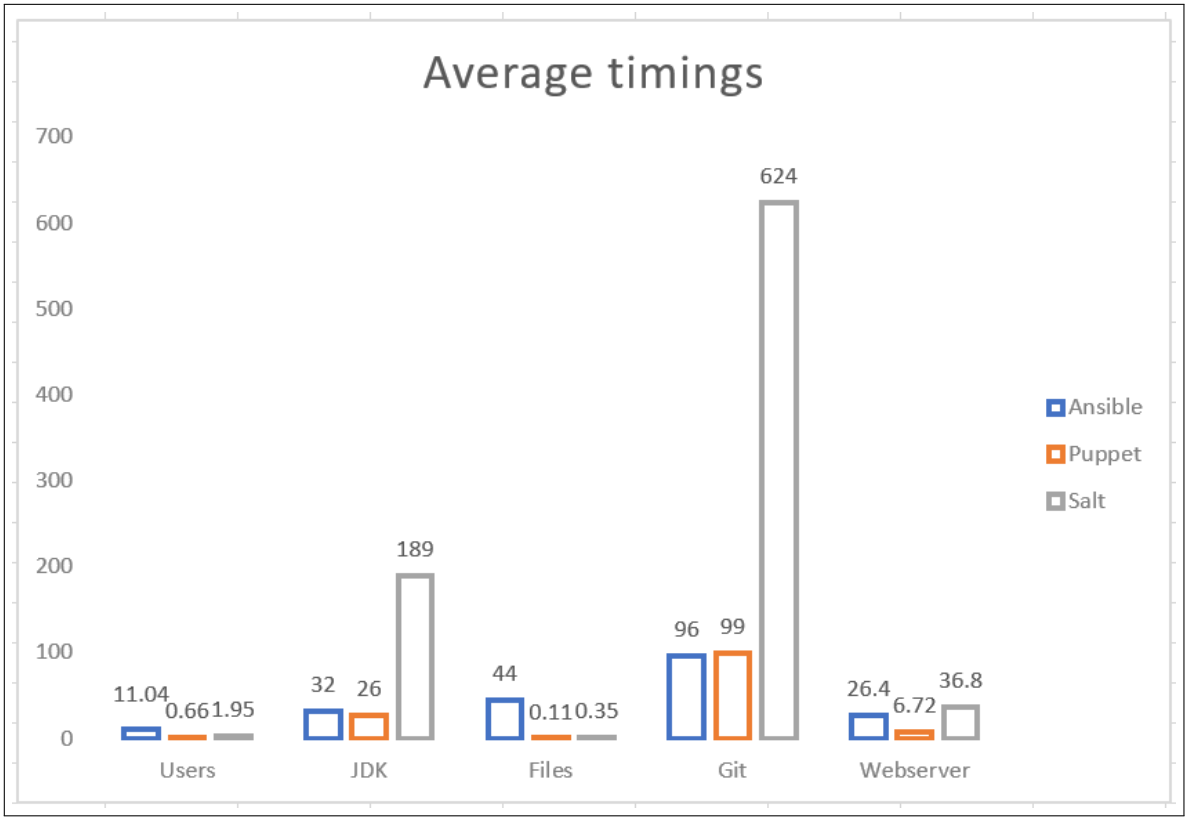

Fig. 6.1 **Average Execution Times by Tool**

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. As the timing was focused on the execution of state application on the slave node, competing with other nodes initially was not a consideration. Assessing the collected results of Ansible and Puppet, this appears to be the case but when examining the large difference in the tool's results, Salt 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.

It's worth noting that 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, which was too challenging to integrate into the framework within the schedule of this research, Puppet was certainly 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 and it's clear from the lack of configuration needed to get Salt worker that the team managing the tool have taken a lot of the work away from the user and automated it. Ansible has tried to be even simpler, needing just the package to be installed on the master node through normal package manager commands to have a working configuration master, with all managed slave nodes simply logged to the /etc/hosts file on said 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. The framework essentially had to double its efforts to manage the tool as part of the testing done for this research.

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), and 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, this is where the learning curve of 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 to successfully install and configure the master software, 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.

6.2 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 and as such, the measurements obtained for it are based on the research done to understand how to integrate it into the framework and therefore are singular run figures. 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 the registration API being called before the user-data within the provisioning code of AWS completes execution, resulting in a dependency for the test not being available. Another example is that Puppet requires host names to be updated and made available to the master and slave nodes. If rapidly creating and destroying resources, sometimes Puppet will look to a terminated EC2 instance with the same host name as the live configuration master. These efforts in increasing the stability are all useful but their Return On Investment (ROI) for the time spent was not deemed high enough for the limited time available for this study.

The effort spent in building the framework also meant that the time allocated to running the tests and gathering metrics was compressed and this could lead to results that are not as representative as they should be. Salt's variance in particular is something that it would be interesting to see how multiples of the test runs executed could modify the average timings

or if it would continue to vary as much as it has in the limited set of testing done as part of this research.

Figure 6.2 shows the different times that testing occurred, and this shows that approximately half of the day had no tests executed during it, and also there is an insight into the number of tests performed as part of this research. Coverage across every hour of the day as well increasing the overall number of tests, from the current number of approximately 11 runs per test per tool, would also grant higher levels of confidence in the metrics gathered and their inferred deductions.

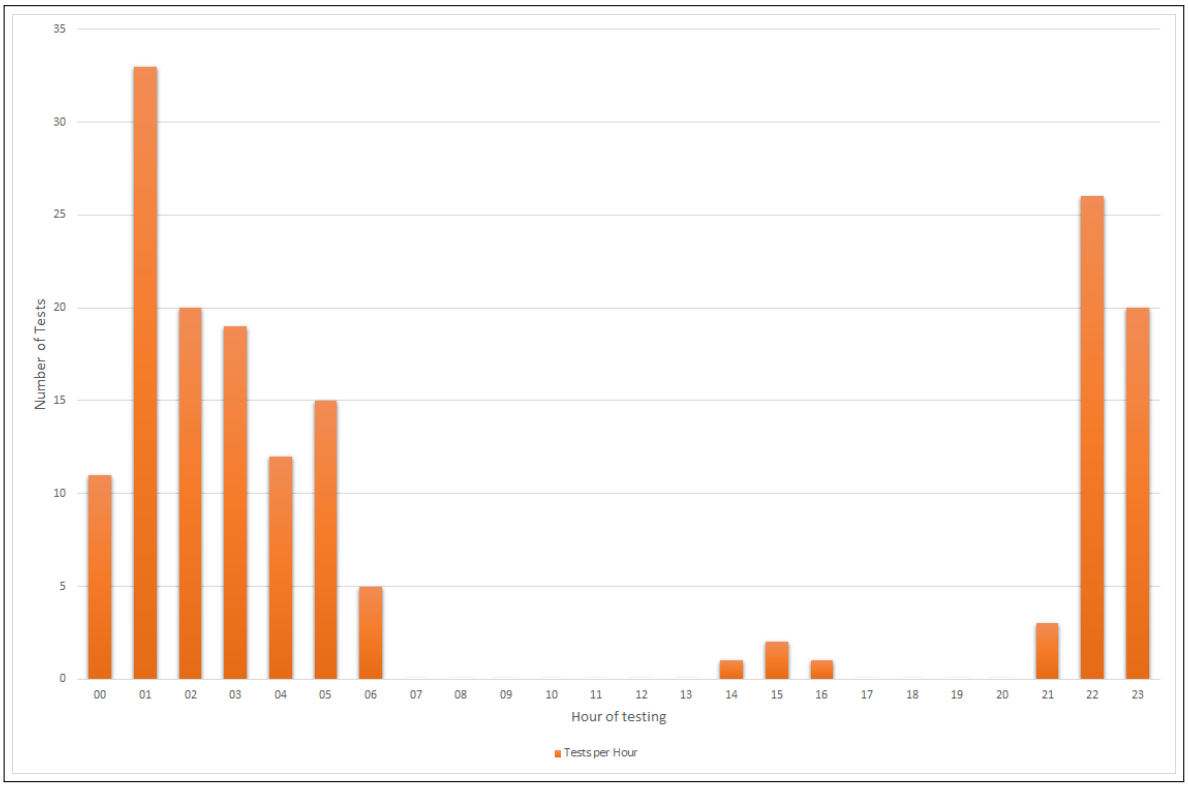

Fig. 6.2 **Breakdown of testing times across this research**

The intention for this framework was to offer a platform agnostic way of obtaining test metrics, but in order to get viable and comparable metrics across the tools, a controlled set of environmental variables were chosen and used. This will have a limiting factor in identifying how these tools work as there are many architectures that are not represented by the choices made in building this framework. To combat this, Ubuntu, one of the most popular and cross platform operating systems was identified and chosen as a platform as well as choosing tests for activities likely to be seen in corporate configuration management, such as creating users, cloning repositories and setting up files and folders ensures that, while we have only a small

subsection of tests that are documented within this study, these tests are activities that are common across most corporate IT infrastructures.

6.3 Future Work

For future work, there are a number of tasks that would be worthy of consideration, some addressing limitations identified in the previous section and others designed to further the study goals and objectives. For the purposes of increasing the velocity of the testing, there are a number improvements that could be made to the framework, primarily in the reporting section. Currently, the raw logs are uploaded to S3 and from there, all log management is manually completed. This is not sustainable for large numbers of test runs as each node under test uploads it's own log file (or two if Puppet is under test) 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. There are more tools that could also be added and from a testing perspective, there is virtually no limit to the the tests that could be added but a more targeted and niche investigation probably makes more sense for enhancement in scope for the future.

Within the existing functionality, if time were available, further enhancements of the Azure terraform code to allow the flexibility to create multiple nodes, compatible with the test framework would aid testing in a way to allow much more efficient result gathering. The ability to spin up multiple nodes is present but to enable it to work with the framework, updates to the naming conventions for the networking components would need to be implemented, along with corresponding updates to the bash scripts responsible for the test execution. 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

Agarwal, A., Gupta, S. and Choudhury, T. (2018) Continuous and integrated software development using devops *2018 International Conference on Advances in Computing and Communication Engineering (ICACCE)*

Amazon (2019) Amazon uptime metrics

Ansible, I. (2019) Ansible galaxy

Basher, M. (2019) Devops: An explorative case study on the challenges and opportunities in implementing infrastructure as code Ph.D. Thesis Umeå University

van der Bent, E., Hage, J., Visser, J. and Gousios, G. (2018) How good is your puppet? an empirically defined and validated quality model for puppet *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*

Chef, L. (2019a) Start improving your chef, automation, and devops skills today - only at learn chef rally!

Chef, S. (2019b) Welcome - the resource for chef cookbooks - chef supermarket

Cowie, J. (2014) *Customizing Chef* O'Reilly 1st ed.

ElFakharany, A. (2018) How to solve "unable to load module" when using paramiko package in a lambda function? - linux school online

Hong, K. (2018) Puppet master post install tasks - 2018

Johari, A. (2019) Chef vs puppet vs ansible vs saltstack: Which one to choose | edureka

Kernal, D. (2019) Set up puppet master and agent on aws ec2

Klein, J. and Reynolds, D. (2019) Infrastructure as code: Final report *Software Engineering Institute*

Lerner, A. (2014) The cost of downtime

O'Connor, R.V., Elger, P. and Clarke, P.M. (2017) Continuous software engineering-a microservices architecture perspective *Journal of Software: Evolution and Process* **29**(11), p. e1866

Oladapo, V. and Onyeaso, G. (2018) Empirical investigation of the moderating effects of organizational size on ecommerce capabilities and organizational performance *International Journal of Economics, Business and Finance* **5**(1), pp. 1–9

Perera, N. and Beck, T. (2018) Continuous delivery: Software deployment and configuration management for critical operations environments *15th International Conference on Space Operations*

Plant, T. (2014) *Downtime costs per industry*

Puppet, F. (2019) Puppet forge

Rahman, A., Mahdavi-Hezaveh, R. and Williams, L. (2019) A systematic mapping study of infrastructure as code research *Information and Software Technology* **108**, pp. 65–77

Salt, F. (2019) Project introduction - saltstack-formulas master documentation

Sandvine (2018) *The Global Internet Phenomena Report*

Scheuner, J. and Leitner, P. (2019) Performance benchmarking of infrastructure-as-a-service (iaas) clouds with cloud workbench *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering - ICPE '19*

Schwarz, J., Steffens, A. and Lichter, H. (2018) Code smells in infrastructure as code *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*

Spadafora, A. (2019) Aws hit by major ddos attack

Thomas, D., Joosen, W. and Vanbrabant, B. (2010) A survey of system configuration tools *Proceedings of the 23rd Large Installations Systems Administration (LISA) conference* pp. 1–14

Wiggins, A. (2017) The twelve-factor app

Williamson, S. (2017) Get the public ip of an azure vm using the azure instance metadata service - northwest cadence

Zhu, L., Xu, D., Xu, X., Tran, A.B., Weber, I. and Bass, L. (2014) Challenges in practicing high frequency releases in cloud environments *Proc. of the Int. Workshop on Release Engineering*

Önnberg, F. (2012) Software configuration management: A comparison of chef, cfengine and puppet

Appendix A

Test Framework Code

A.1 Terraform

The following is the code used in the Terraform files for recreation of the framework.

A.1.1 Configuration Master file

```
provider "aws" {
  region = "eu-west-1"
}

variable "configTool" {
  default = "puppet"    # salt, puppet, ansible
}

locals  {
  instance-userdata3 = <<EOF
#! /bin/bash -x
date >> /provisionedAt.txt
sudo apt-get update
sudo apt-get -y install awscli > /output.log 2>&1
git clone https://github.com/RedXIV2/terraform.git
sudo bash /terraform/scripts/setupBaseFiles.sh "${var.configTool}"
ls /tmp >> /didKeyArrive.txt
echo '# Check if the ssh-agent is already running' >> /home/ubuntu/.bashrc
echo 'if [[ "$(ps -u $USER | grep ssh-agent | wc -l)" -lt "1" ]]; then' >>
 ↪  /home/ubuntu/.bashrc
```

```
echo '    #echo "$(date +%F@%T) - SSH-AGENT: Agent will be started"' >>
 ↪  /home/ubuntu/.bashrc
echo '  # Start the ssh-agent and redirect the environment variables into a file'
 ↪  >> /home/ubuntu/.bashrc
echo '    ssh-agent -s >~/.ssh/ssh-agent' >> /home/ubuntu/.bashrc
echo '    # Load the environment variables from the file' >> /home/ubuntu/.bashrc
echo '    . ~/.ssh/ssh-agent >/dev/null' >> /home/ubuntu/.bashrc
echo '    # Add the default key to the ssh-agent' >> /home/ubuntu/.bashrc
echo '    chmod 400 /tmp/awsthesis.pem' >> /home/ubuntu/.bashrc
echo '    ssh-add /tmp/awsthesis.pem' >> /home/ubuntu/.bashrc
echo 'else' >> /home/ubuntu/.bashrc
echo '    #echo "$(date +%F@%T) - SSH-AGENT: Agent already running"' >>
 ↪  /home/ubuntu/.bashrc
echo '    . ~/.ssh/ssh-agent >/dev/null' >> /home/ubuntu/.bashrc
echo 'fi' >> /home/ubuntu/.bashrc
  EOF
  }

data "aws_ami" "amazon_linux" {
  most_recent = true

  filter {
    name   = "name"
    values = ["amzn-ami-hvm-*-x86_64-gp2"]
  }

  owners = ["amazon"]
}

data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"]
  }
owners = ["099720109477"]
}

resource "aws_security_group" "ingress-all-test" {
name = "allow-all-sg"

ingress {
    cidr_blocks = [
```

```
      "0.0.0.0/0"
    ]
from_port = 0
    to_port = 0
    protocol = "-1"

  }
// Terraform removes the default rule
  egress {
   from_port = 0
   to_port = 0
   protocol = "-1"
   cidr_blocks = ["0.0.0.0/0"]
 }

}
resource "aws_instance" "web" {
  ami               = "${data.aws_ami.ubuntu.id}"
  instance_type     =  "t2.medium"
  key_name          = "awsthesis"
  iam_instance_profile  = "configMaster"
  security_groups = ["${aws_security_group.ingress-all-test.name}"]

  tags = {
    Name = "ConfigMaster"
  }

  connection {
    host = "${aws_instance.web.public_ip}"
    type = "ssh"
    user = "ubuntu"
    private_key = "${file("D:\\Tools\\Keys\\awsthesis.pem")}"
    agent = false
  }

  provisioner "file" {
    source      = "D:\\Tools\\Keys\\awsthesis.pem"
    destination = "/tmp/awsthesis.pem"
  }

  user_data = "${local.instance-userdata}"
}
```

A.1.2 AWS Slave Node

```
provider "aws" {
  region = "eu-west-1"
}

locals  {
  instance-userdata2 = <<EOF
#! /bin/bash
date >> provisionedAt.txt
EOF
  }

locals  {
  instance-userdata = <<EOF
#! /bin/bash
date >> provisionedAt.txt
sudo apt install -y awscli
sudo apt install -y python
EOF
  }

data "aws_ami" "amazon_linux" {
  most_recent = true

  filter {
    name   = "name"
    values = ["amzn-ami-hvm-*-x86_64-gp2"]
  }

  owners = ["amazon"]
}

data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"]
  }
owners = ["099720109477"]
}
```

```
resource "aws_instance" "web" {
count = 3

  ami               = "${data.aws_ami.ubuntu.id}"
  instance_type     = "t2.micro"
  key_name          = "awsthesis"
  user_data_base64  = "${base64encode(local.instance-userdata)}"
  iam_instance_profile  = "configMaster"

  tags = {
    Name = "test-server-${count.index}"
  }

  connection {
    host = "${self.public_ip}"
    type = "ssh"
    user = "ubuntu"
    private_key = "${file("D:\\Tools\\Keys\\awsthesis.pem")}"
    agent = false
  }

  provisioner "remote-exec" {
      inline = [
        "sudp apt-get update",
        "sudo git clone https://github.com/RedXIV2/terraform.git /terraform",
        "sudo cp /terraform/scripts/addToFamily/addTo* /",
        "sudo chmod 777 /addTo*",
        "curl --retry 5 -m 120 -X GET
         ↪  '${var.registrationAPI}?ipAddress=${self.private_ip}&cmTool=Salt&testSuite=2&platform
      ]
  }
}
```

A.1.3 Azure Slave Node

```
provider "azurerm" {
  tenant_id       = "${var.tenant_id}"
  subscription_id = "${var.subscription_id}"
  client_id       = "${var.client_id}"
  client_secret   = "${var.client_secret}"
}
```

```
data "azurerm_resource_group" "resource_group" {
  name                = "thesisGroup"
}

# Create virtual network
resource "azurerm_virtual_network" "myterraformnetwork" {
    count = 1
    name                = "myVnet"
    address_space       = ["10.0.0.0/16"]
    location            = "northeurope"
    resource_group_name = "${data.azurerm_resource_group.resource_group.name}"

    tags = {
        environment = "Terraform Demo"
    }
}

# Create subnet
resource "azurerm_subnet" "myterraformsubnet" {
    count = 1
    name                 = "mySubnet"
    resource_group_name  = "${data.azurerm_resource_group.resource_group.name}"
    virtual_network_name =
     ↪  "${azurerm_virtual_network.myterraformnetwork[count.index].name}"
    address_prefix       = "10.0.1.0/24"
}

# Create public IPs
resource "azurerm_public_ip" "myterraformpublicip" {
count = 1
    name                         = "myPublicIP"
    location                     = "northeurope"
    resource_group_name          =
     ↪  "${data.azurerm_resource_group.resource_group.name}"
    allocation_method            = "Dynamic"
    domain_name_label            = "thesisnode"

    tags = {
        environment = "Terraform Demo"
    }
}

# Create Network Security Group and rule
```

```
resource "azurerm_network_security_group" "myterraformnsg" {
    count = 1
    name                = "myNetworkSecurityGroup"
    location            = "northeurope"
    resource_group_name = "${data.azurerm_resource_group.resource_group.name}"

    security_rule {
        name                       = "SSH"
        priority                   = 1001
        direction                  = "Inbound"
        access                     = "Allow"
        protocol                   = "Tcp"
        source_port_range          = "*"
        destination_port_range     = "22"
        source_address_prefix      = "*"
        destination_address_prefix = "*"
    }

    tags = {
        environment = "Terraform Demo"
    }
}

# Create network interface
resource "azurerm_network_interface" "myterraformnic" {
    count = 1
    name                      = "myNIC"
    location                  = "northeurope"
    resource_group_name       =
     ↪  "${data.azurerm_resource_group.resource_group.name}"
    network_security_group_id =
     ↪  "${azurerm_network_security_group.myterraformnsg[count.index].id}"

    ip_configuration {
        name                          = "myNicConfiguration"
        subnet_id                     =
         ↪  "${azurerm_subnet.myterraformsubnet[count.index].id}"
        private_ip_address_allocation = "Dynamic"
        public_ip_address_id          =
         ↪  "${azurerm_public_ip.myterraformpublicip[count.index].id}"
    }

    tags = {
        environment = "Terraform Demo"
```

```
    }
}

# Generate random text for a unique storage account name
resource "random_id" "randomId" {
    count = 1
    keepers = {
        # Generate a new ID only when a new resource group is defined
        resource_group = "${data.azurerm_resource_group.resource_group.name}"
    }

    byte_length = 8
}

# Create storage account for boot diagnostics
resource "azurerm_storage_account" "mystorageaccount" {
    count = 1
    name                        = "diag${random_id.randomId[count.index].hex}"
    resource_group_name         =
     ↪  "${data.azurerm_resource_group.resource_group.name}"
    location                    = "northeurope"
    account_tier                = "Standard"
    account_replication_type    = "LRS"

    tags = {
        environment = "Terraform Demo"
    }
}

# Create virtual machine
resource "azurerm_virtual_machine" "myterraformvm" {
count = 1

    name                          = "myVM-${count.index}"
    location                      = "northeurope"
    resource_group_name           =
     ↪  "${data.azurerm_resource_group.resource_group.name}"
    network_interface_ids         =
     ↪  ["${azurerm_network_interface.myterraformnic[count.index].id}"]
    vm_size                       = "Standard_DS1_v2"
    delete_os_disk_on_termination = "true"

    storage_os_disk {
        name              = "myOsDisk"
```

```
    caching           = "ReadWrite"
    create_option     = "FromImage"
    managed_disk_type = "Premium_LRS"
}

storage_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "16.04.0-LTS"
    version   = "latest"
}

os_profile {
    computer_name  = "myvm"
    admin_username = "ubuntu"
}

os_profile_linux_config {
    disable_password_authentication = true
    ssh_keys {
        path     = "/home/ubuntu/.ssh/authorized_keys"
        key_data = "ssh-rsa
         ↪  AAAAB3NzaC1yc2EAAAABJQAAAQEAgXBfFArZjcOiDZy25jG/fOBy3NgDZAWBdBSZUtXGIARrWHahCGVe8
    }
}

boot_diagnostics {
    enabled = "true"
    storage_uri =
     ↪  "${azurerm_storage_account.mystorageaccount[count.index].primary_blob_endpoint}"
}

tags = {
    environment = "Terraform Demo"
}

connection {
  host        = "thesisnode.northeurope.cloudapp.azure.com"
  type        = "ssh"
  user        = "ubuntu"
  private_key = "${file("D:\\Tools\\Keys\\azureThesis.pem")}"
  timeout     = "1m"
}
```

```
    provisioner "remote-exec" {

      inline = [
          "sleep 20",
          "date >> provisionedAt.txt",
          "sudo mkdir /home/ubuntu/.aws",
          "sudo echo [default] >> /home/ubuntu/credentials",
          "sudo echo aws_access_key_id = ${var.aws_access_id} >>
           ↪  /home/ubuntu/credentials",
          "sudo echo aws_secret_access_key =${var.aws_secret_id} >>
           ↪  /home/ubuntu/credentials",
          "sudo echo [default] >> /home/ubuntu/config",
          "sudo echo output = json >> /home/ubuntu/config",
          "sudo echo region = eu-west-1 >> /home/ubuntu/config",
          "sudo cp /home/ubuntu/credentials /home/ubuntu/.aws/",
          "sudo cp /home/ubuntu/config /home/ubuntu/.aws/",
          "sudo git clone https://github.com/RedXIV2/terraform.git /terraform",
          "sudo cp /terraform/scripts/addToFamily/addTo* /",
          "sudo chmod 777 /addTo*",
          "sudo echo ${var.key_data} >> /home/ubuntu/.ssh/authorized_keys",
          "export myIP=$(curl -H Metadata:true
           ↪  \"http://169.254.169.254/metadata/instance/network/interface/0/ipv4/ipAddress/0/public
          "curl --retry 5 -m 120 -X GET
           ↪  \"${var.registrationAPI}?ipAddress=$myIP&cmTool=Puppet&testSuite=5&platform=azure\""

    ]
  }
}

data "azurerm_public_ip" "myterraformpublicip" {
count = 1
  name                =
   ↪  "${azurerm_public_ip.myterraformpublicip[count.index].name}"
  resource_group_name =
   ↪  "${azurerm_virtual_machine.myterraformvm[count.index].resource_group_name}"
}

output "IPAddress" {
  value = "${data.azurerm_public_ip.myterraformpublicip.*.ip_address}"
}
```

A.2 Lambda Code

```
# -*- coding: utf-8 -*-
#
# Author: Dave Hill
# This is used for new EC2 instances to connect to a configuration master

import json
import boto3
import paramiko

def lambda_handler(event, context):

    hostIP = event['ipAddress']
    cmTool = event['cmTool']
    testSuite = event['testSuite']
    platform = event['platform']

# get the config master IP
    ec2 = boto3.client('ec2')
    machines = ec2.describe_instances(Filters=[{'Name': 'tag:Name', 'Values':
     ↪  ['ConfigMaster']}])
    for reservation in machines["Reservations"]:
        for instance in reservation["Instances"]:
            if instance["PublicDnsName"]:
                configMasterIP = instance["PublicDnsName"]

    if cmTool == "Salt":
        if platform == "aws":
            configMasterIP = hostIP
            machines = ec2.describe_instances(Filters=[{'Name':
             ↪  'private-ip-address', 'Values': [hostIP]}])
            for reservation in machines["Reservations"]:
                for instance in reservation["Instances"]:
                    if instance["PublicDnsName"]:
                        configMasterIP = instance["PublicDnsName"]
        else:
            configMasterIP = "thesisnode.northeurope.cloudapp.azure.com"

    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    privkey = paramiko.RSAKey.from_private_key_file('awsthesis.pem')
```

```
ssh.connect(
    configMasterIP, username='ubuntu', pkey=privkey
)

print(configMasterIP)
if platform == "azure":
    commandToRun = "sudo -b bash /addTo{}.sh {} {} {} > /dev/null 2>&1
     ↪  &".format(cmTool, hostIP, testSuite, platform)
else:
    commandToRun = "sudo -b bash /addTo{}.sh {} {} > /dev/null 2>&1
     ↪  &".format(cmTool, hostIP, testSuite)

print(commandToRun)

ssh.exec_command(commandToRun)
ssh.close()

if cmTool == "Puppet":
    puppetClientIP = hostIP
    machines = ec2.describe_instances(Filters=[{'Name': 'private-ip-address',
     ↪  'Values': [hostIP]}])
    for reservation in machines["Reservations"]:
        for instance in reservation["Instances"]:
            if instance["PublicDnsName"]:
                puppetClientIP = instance["PublicDnsName"]
    ssh2 = paramiko.SSHClient()
    ssh2.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh2.connect(
        puppetClientIP, username='ubuntu', pkey=privkey
    )
    if platform == "azure":
        commandToRun2 = "sudo -b bash /addTo{}Client.sh {} {} {} > /dev/null
         ↪  2>&1 &".format(cmTool, hostIP, testSuite, platform)
    else:
        commandToRun2 = "sudo -b bash /addTo{}Client.sh {} {} > /dev/null 2>&1
         ↪  &".format(cmTool, hostIP, testSuite)
    ssh2.exec_command(commandToRun2)
    ssh2.close()

return {
    'statusCode': 200,
    'body': json.dumps('Node Registered')
}
```

A.3 Bash File Library

A.3.1 SetupBaseFiles.sh

```
# Create the log file for all future system logging
touch /myLogs.txt
chmod 777 /myLogs.txt
echo "$(date) Log File created and permissions set" >> myLogs.txt

# copy addTo Family of scripts to / directory
cp /terraform/scripts/addToFamily/addTo* /
chmod 700 /addTo*
echo "$(date) AddToFamily scripts copied and permissions set" >> myLogs.txt

# change key permissions
chmod 600 /tmp/awsthesis.pem
echo "$(date) key permissions updated"

#set up config master server with Ansible
if [ "$1" == "ansible" ]
then
sudo apt-add-repository --yes --update ppa:ansible/ansible
sudo apt install -y ansible >> /output.log 2>&1

sudo sed -i '/callback_whitelist/c\callback_whitelist = profile_tasks'
 ↪  /etc/ansible/ansible.cfg
sudo sed -i '/host_key_checking/c\host_key_checking = False'
 ↪  /etc/ansible/ansible.cfg
sudo sed -i '/#remote_user/c\remote_user = ubuntu' /etc/ansible/ansible.cfg
sudo sed -i '/#private_key_file/c\private_key_file = /tmp/awsthesis.pem'
 ↪  /etc/ansible/ansible.cfg
fi

#set up config master server with Salt
if [ "$1" == "salt" ]
then
wget -O -
 ↪  https://repo.saltstack.com/apt/ubuntu/16.04/amd64/latest/SALTSTACK-GPG-KEY.pub
 ↪  | sudo apt-key add -
sudo touch /etc/apt/sources.list.d/saltstack.list
sudo echo "deb http://repo.saltstack.com/apt/ubuntu/16.04/amd64/latest xenial main"
 ↪  >> /etc/apt/sources.list.d/saltstack.list
```

```
sudo apt-get install -y salt-master
sudo apt-get install -y salt-minion
sudo apt-get install -y salt-ssh
sudo apt-get install -y salt-syndic
sudo apt-get install -y salt-cloud
sudo apt-get install -y salt-api

sudo echo "auto_accept: True" >> /etc/salt/master
sudo echo "file_roots: " >> /etc/salt/master
sudo echo "  base: " >> /etc/salt/master
sudo echo "    - /srv/salt " >> /etc/salt/master

sudo service salt-master start

sudo mkdir /srv/salt
sudo cp /terraform/tests/Salt/top.sls /srv/salt/
sudo cp /terraform/tests/Salt/*/*.sls /srv/salt/

sudo echo "while true; do" >> approveAll.sh
sudo echo "  sudo salt-key -y -A" >> approveAll.sh
sudo echo "  sleep 1;" >> approveAll.sh
sudo echo "done" >> approveAll.sh

chmod +x approveAll.sh
bash approveAll.sh &

fi

#set up config master server with Puppet
if [ "$1" == "puppet" ]
then

sudo hostnamectl set-hostname puppet-master
sudo apt update -y
sudo apt upgrade -y

sudo apt install puppetmaster -y
sudo echo "autosign=true" >> /etc/puppet/puppet.conf

#needed for JDK test
sudo puppet module install example42-java --version 2.1.2
#needed for git test
sudo puppet module install puppetlabs-git --version 0.5.0
```

```
#needed for apache test
sudo puppet module install example42-apache --version 2.1.13

sudo mkdir -p /etc/puppet/modules/myuser/manifests/
sudo cp /terraform/tests/Puppet/init.pp /etc/puppet/modules/myuser/manifests/

fi
```

A.3.2 addToAnsible.sh

```
if [ -z "$1" ]
  then
    echo "$(date) No argument supplied" >> /myLogs.txt
    exit 1
fi

if grep -q "$1" /etc/ansible/hosts; then
    echo "$(date) $1 already in file" >> /myLogs.txt
    exit 0
fi

if grep -q "\[servers\]" /etc/ansible/hosts; then
    echo "$(date) found and added $1" >> /myLogs.txt
    echo "$1" >> /etc/ansible/hosts
else
    echo "$(date) not found and added $1" >> /myLogs.txt
    echo [servers] >> /etc/ansible/hosts
    echo "$1" >> /etc/ansible/hosts
fi

echo "$(date) running sudo bash /terraform/scripts/runTests.sh $2 Ansible" >>
 ↪  /myLogs.txt
sudo bash /terraform/scripts/runTests.sh $2 Ansible $1
```

A.3.3 addToSalt.sh

```
if [ -z "$1" ]
  then
    echo "$(date) No argument supplied" >> /myLogs.txt
```

```
    exit 1
fi

echo "$(date) installing AWSCLI" >> /myLogs.txt
sudo apt update
sudo apt install -y awscli

echo "$(date) checking AWSCLI" >> /myLogs.txt
until [ -x "$(command -v aws)" ]
do
  echo 'Error: aws is not installed.' >&2
  sudo apt install -y awscli
done


echo "$(date) Searching for config master ..." >> /myLogs.txt

CONFIG_DNS=$(sudo aws ec2 describe-instances --filters
 ↪  'Name=tag:Name,Values=ConfigMaster' \
  'Name=instance-state-name,Values=running' --query
   ↪  'Reservations[*].Instances[*].[PrivateDnsName]' \
   --region eu-west-1 --output text)

PUBLIC_CONFIG_DNS=$(sudo aws ec2 describe-instances --filters
 ↪  'Name=tag:Name,Values=ConfigMaster' \
  'Name=instance-state-name,Values=running' --query
   ↪  'Reservations[*].Instances[*].[PublicDnsName]' \
   --region eu-west-1 --output text)

echo "$(date) Config DNS is ${CONFIG_DNS}" >> /myLogs.txt
echo "$(date) Config DNS is ${PUBLIC_CONFIG_DNS}" >> /myLogs.txt

echo "$(date) Getting bootstrap script" >> /myLogs.txt
sudo curl -o bootstrap-salt.sh -L https://bootstrap.saltstack.com

echo "$(date) bootstrap needs to be executable" >> /myLogs.txt
sudo chmod +x bootstrap-salt.sh

echo "$(date) bootstrapping salt minion..." >> /myLogs.txt
if [ -z "$3" ]
  then
    sudo sh bootstrap-salt.sh -A "${CONFIG_DNS}"
  else
    sudo sh bootstrap-salt.sh -A "${PUBLIC_CONFIG_DNS}"
```

```
fi

echo "$(date) Bootstrapping complete" >> /myLogs.txt


echo "$(date) running sudo bash /terraform/scripts/runTests.sh $2 Salt" >>
 ↪  /myLogs.txt
sudo bash /terraform/scripts/runTests.sh $2 Salt $1
```

A.3.4 addToPuppet.sh

```
if [ -z "$1" ]
  then
    echo "$(date) No argument supplied" >> /myLogs.txt
    exit 1
fi

echo "$(date) installing AWSCLI" >> /myLogs.txt
sudo apt update
sudo apt install -y awscli

echo "$(date) checking AWSCLI" >> /myLogs.txt
until [ -x "$(command -v aws)" ]
do
  echo 'Error: aws is not installed.' >&2
  sudo apt install -y awscli
done

echo "$(date) Searching for client details ..." >> /myLogs.txt

CONFIG_DNS="$(sudo aws ec2 describe-instances --filters
 ↪  "Name=private-ip-address,Values=$1" \
  'Name=instance-state-name,Values=running' --query
   ↪  'Reservations[*].Instances[*].[PrivateDnsName]' \
   --region eu-west-1 --output text)"

echo "$(date) Private DNS is ${CONFIG_DNS}" >> /myLogs.txt

PUBLIC_DNS="$(sudo aws ec2 describe-instances --filters
 ↪  "Name=private-ip-address,Values=$1" \
  'Name=instance-state-name,Values=running' --query
   ↪  'Reservations[*].Instances[*].[PublicDnsName]' \
```

```
   --region eu-west-1 --output text)"

echo "$(date) Public DNS is ${PUBLIC_DNS}" >> /myLogs.txt

NODE_ID="$(sudo aws ec2 describe-instances --filters
 ↪  "Name=private-ip-address,Values=$1" \
  'Name=instance-state-name,Values=running' --query
   ↪  'Reservations[*].Instances[*].Tags[*].Value' \
   --region eu-west-1 --output text)"

NODE_ID_NUMBER=$(echo ${NODE_ID: -1} )

echo "$(date) Node Identifier is ${NODE_ID_NUMBER}" >> /myLogs.txt
if [ -z "$3" ]
  then
  HOST_ENTRY="$1 puppet-agent-$NODE_ID_NUMBER $CONFIG_DNS $PUBLIC_DNS
   ↪  puppet-agent-$NODE_ID_NUMBER.eu-west-1.compute.internal"

  echo "$(date) Adding $HOST_ENTRY to hosts file" >> /myLogs.txt
  sudo echo $HOST_ENTRY >> /etc/hosts
  else
  HOST_ENTRY="$1 thesisnode.northeurope.cloudapp.azure.com"
  echo "$(date) Adding $HOST_ENTRY to hosts file" >> /myLogs.txt
  sudo echo $HOST_ENTRY >> /etc/hosts
fi

sudo service puppetmaster restart

echo "$(date) running sudo bash /terraform/scripts/runTests.sh $2 Puppet" >>
 ↪  /myLogs.txt
sudo bash /terraform/scripts/runTests.sh $2 Puppet $1
```

A.3.5 addToPuppetClient.sh

```
if [ -z "$1" ]
  then
    echo "$(date) No argument supplied" >> /myLogs.txt
    exit 1
fi
```

```
echo "$(date) installing AWSCLI" >> /myLogs.txt
sudo apt update -y
sudo apt upgrade -y
sudo apt install -y awscli

echo "$(date) checking AWSCLI" >> /myLogs.txt
until [ -x "$(command -v aws)" ]
do
  echo 'Error: aws is not installed.' >&2
  sudo apt-get update
  sudo apt install -y awscli
done

NODE_ID=$(sudo aws ec2 describe-instances --filters
 ↪  "Name=private-ip-address,Values=$1" \
  'Name=instance-state-name,Values=running' --query
   ↪  'Reservations[*].Instances[*].Tags[*].Value' \
   --region eu-west-1 --output text)

NODE_ID_NUMBER=$(echo ${NODE_ID: -1} )

sudo hostnamectl set-hostname puppet-agent-"${NODE_ID_NUMBER}"


echo "$(date) Searching for config master ..." >> /myLogs.txt

CONFIG_DNS=$(sudo aws ec2 describe-instances --filters
 ↪  'Name=tag:Name,Values=ConfigMaster' \
  'Name=instance-state-name,Values=running' --query
   ↪  'Reservations[*].Instances[*].[PrivateDnsName]' \
   --region eu-west-1 --output text)

echo "$(date) Private DNS is ${CONFIG_DNS}" >> /myLogs.txt

PUBLIC_DNS=$(sudo aws ec2 describe-instances --filters
 ↪  'Name=tag:Name,Values=ConfigMaster' \
  'Name=instance-state-name,Values=running' --query
   ↪  'Reservations[*].Instances[*].[PublicDnsName]' \
   --region eu-west-1 --output text)

echo "$(date) Config DNS is ${PUBLIC_DNS}" >> /myLogs.txt
```

```
PRIVATE_IP=$(sudo aws ec2 describe-instances --filters
 ↪  'Name=tag:Name,Values=ConfigMaster' \
  'Name=instance-state-name,Values=running' --query
   ↪  'Reservations[*].Instances[*].[PrivateIpAddress]' \
   --region eu-west-1 --output text)

PUBLIC_IP=$(sudo aws ec2 describe-instances --filters
 ↪  'Name=tag:Name,Values=ConfigMaster' \
  'Name=instance-state-name,Values=running' --query
   ↪  'Reservations[*].Instances[*].[PublicIpAddress]' \
   --region eu-west-1 --output text)

echo "$(date) Private IP is ${PRIVATE_IP}" >> /myLogs.txt

if [ -z "$3" ]
  then
  HOST_ENTRY="$PRIVATE_IP puppet-master $CONFIG_DNS $PUBLIC_DNS
   ↪  puppet-master.eu-west-1.compute.internal"
  else
  HOST_ENTRY="$PUBLIC_IP puppet-master puppet-master.eu-west-1.compute.internal"
fi

echo "$(date) Adding $HOST_ENTRY to hosts file"
sudo echo $HOST_ENTRY >> /etc/hosts

until [ -x "$(command -v puppet)" ]
do
  sudo apt install puppet -y >> /myLogs.txt
done

echo "$(date) Updating puppet.conf" >> /myLogs.txt
sudo sed -i '/postrun_command=\/etc\/puppet\/etckeeper-commit-post/a server =
 ↪  puppet-master.eu-west-1.compute.internal' /etc/puppet/puppet.conf

sudo puppet agent --no-daemonize --onetime --verbose >> /myLogs-$1.txt
sudo puppet agent --enable >> /myLogs-$1.txt

sudo puppet agent --server puppet-master.eu-west-1.compute.internal >>
 ↪  /myLogs-$1.txt


echo "$(date) Applying Puppet State" >> /myLogs-$1.txt
sudo puppet agent --test >> /myLogs-$1.txt
```

```
echo "$(date) Executing: aws s3 mv /myLogs-$1.txt
 ↪  s3://dhill-config-management-tests/testResults/${2}/myLogs-$(date).txt >>
 ↪  awsCopy.log 2>&1" >> /myLogs-$1.txt

aws s3 mv /myLogs-$1.txt
 ↪  s3://dhill-config-management-tests/testResults/Puppet/puppet-client-myLogs-$1-"$(date)".txt
 ↪  >> awsCopy.log 2>&1

echo "$(date) ***Finished Upload***" >> /myLogs.txt
```

A.3.6 runTests.sh

```
if [ -z "$1" ]
  then
    echo "$(date) No testcase identifier supplied" >> /myLogs-$3.txt
    exit 1
fi

if [ -z "$2" ]
  then
    echo "$(date) No testcase technology supplied" >> /myLogs-$3.txt
    exit 1
fi

if [ -z "$3" ]
  then
    echo "$(date) No IP address supplied" >> /myLogs-$3.txt
    exit 1
fi

cat /provisionedAt.txt >> /myLogs-$3.txt

echo "$(date) ***Running Test***" >> /myLogs-$3.txt

path_to_test=/terraform/tests/$2/
test_case_to_run=$1
full_test="$(ls ${path_to_test}${test_case_to_run}*/* )"

echo "$(date) Executing: ${full_test}" >> /myLogs-$3.txt
```

```
#Ansible specific test runner
if [ "$2" == "Ansible" ]
then
sudo -u ubuntu ansible-playbook -i $3, ${full_test} >> /myLogs-$3.txt 2>&1
fi

#Salt specific test runner
if [ "$2" == "Salt" ]
then

state_to_apply="$(echo ${full_test} | sed 's|.*/||' | sed 's/.\{4\}$//')"

until [ -x "$(command -v salt-call)" ]
do
  echo 'Error: salt is not installed.' >> /myLogs-$3.txt
done

sudo salt-call state.sls ${state_to_apply} >> /myLogs-$3.txt 2>&1

fi

#Puppet specific test runner
if [ "$2" == "Puppet" ]
then

echo "$(date) copying ${full_test} to /etc/puppet/manifest" >> /myLogs-$3.txt
sudo cp ${full_test} /etc/puppet/manifests/

fi

echo "$(date) ***Finished Test***" >> /myLogs-$3.txt

echo "$(date) ***Uploading Test Results***" >> /myLogs-$3.txt
echo "$(date) Executing: aws s3 mv /myLogs-$3.txt
 ↪  s3://dhill-config-management-tests/testResults/${2}/myLogs-$(date).txt >>
 ↪  awsCopy.log 2>&1" >> /myLogs-$3.txt

aws s3 mv /myLogs-$3.txt
 ↪  s3://dhill-config-management-tests/testResults/"${2}"/myLogs-$3-"$(date)".txt
 ↪  >> awsCopy.log 2>&1

echo "$(date) ***Finished Upload***" >> /myLogs-$3.txt
```