Quality Assurance for Open Source Software Configuration Management

Stefan Meyer∗, Philip Healy∗, Theo Lynn†, John Morrison∗

Irish Centre for Cloud Computing & Commerce
∗University College Cork, Ireland
{s.meyer, p.healy, j.morrison}@cs.ucc.ie
†Dublin City University, Ireland
theo.lynn@dcu.ie

Abstract—Commonly used open source configuration management systems, such as Puppet, Chef and CFEngine, allow for system configurations to be expressed as scripts. A number of quality issues that may arise when executing these scripts are identified. An automated quality assurance service is proposed that identifies the presence of these issues by automatically executing scripts across a range of environments. Test results are automatically published to a format capable of being consumed by script catalogues and social coding sites. This would serve as an independent signal of script trustworthiness and quality to script consumers and would allow developers to be made quickly aware of quality issues. As a result, potential consumers of scripts can be assured that a script is likely to work when applied to their particular environment. Script developers can be notified of compatibility issues and take steps to address them.

Keywords—Automated configuration, configuration management, continuous integration, automated deployment, service orchestration, assurance.

I. INTRODUCTION

Automated configuration management (CM) tools allow IT equipment to be placed in well-defined configurations without direct human intervention. This ability provides a number of benefits in terms of efficiency, scalability and knowledge management. Section II provides a brief introduction to automated configuration management and the rationale behind it, including a brief overview of the leading open source tools (Puppet, Chef and CFEngine).

Automated configuration actions are typically expressed as declarative scripts. As the leading open source CM tools are available for a number of platforms, scripts may end up being executed across a range of environments with differing operating system families, operating system versions, configuration tool versions, and a number of other parameters. Platform-specific details such as file locations and package names can find their way into scripts, leading to execution failures on platforms where these details differ. These failure modes are identified and examined in detail in Section III. Typically, portability issues can be resolved by testing scripts on a variety of platforms and adding platform-specific branches as necessary. However, script developers often neglect to test their scripts on platforms other than those that they deploy to within their own organisations.

Open source social coding communities, such as GitHub [1], are often used to share scripts for open source CM tools and encourage collaborative development. However, consumers have no guarantee that scripts obtained via these communities have been tested on the platform that they wish to deploy to. As a result, scripts may fail to function correctly when deployed and must be altered to rectify platform-specific assumptions. Several scripts with the same functionality may be available. So, consumers may be faced with a choice of multiple implementations, none of which is guaranteed to work when applied to their target platform due to compatibility issues or unmentioned dependencies.

Some tool-specific automated testing platforms are available. However, these do not test for all failure modes and are not used consistently by script developers. We make a case for an independent automated quality assurance platform that exercises software configuration management scripts across a broad variety of tool versions and platforms. Existing tools, such as syntax checkers, are leveraged where appropriate. Virtual machines obtained via Infrastructure-as-a-Service are used to test across heterogeneous platforms. The output of each QA iteration is a list of test results indicating execution success or failure across a variety of scenarios. Multiple open source CM tools are supported. An overview of the proposed platform and a technical sketch of an implementation is provided in Section IV.

Increasingly, CM functionality is being integrated with the automated deployment of required resources via Infrastructure-as-a-Service. This technique is sometimes referred to as service orchestration [2]. Automated deployment introduces additional portability issues as scripts may not function correctly when targeted towards a different cloud provider, or even different regions of the same provider. Section V considers these issues and proposes how they may be addressed by extending the QA platform to automatically test scripts across multiple IaaS endpoints.

An overview of the current systems for quality assurance of CM scripts is provided in Section VI. Related academic work is also considered. Finally, our conclusions are presented in Section VII.
II. CONFIGURATION MANAGEMENT AND AUTODEPLOYMENT

Traditional CM is defined by four related activities: configuration identification, change control, status accounting and configuration audit [3]. Although the importance of these activities has long been recognised, in recent years their application is becoming more widespread as tools that enable their automation have become available. These automated CM tools operate by allowing configuration actions to be specified using a high-level declarative language and stored on a central server. Client machines can compare their current configuration state to the central configuration specifications that apply to them. Configuration actions are applied as necessary to bridge the gap between a system’s current configuration and the desired configuration specified centrally.

Configuration parameters typically include software packages and configuration files for system services. More complex configuration actions include setting passwords, installing security certificates, integration between services, and implementation of a backup policy. From an administration perspective, the main benefits of CM are reliable repetitiveness leading to time savings and reduced scope for human error. The declarative nature of high-level CM also allows for the scalable automated configuration of large numbers of machines, eliminating human administration as a bottleneck to the rapid deployment and reconfiguration of bulk computing resources. From an organisational perspective, the primary benefits of automated CM is the existence of a well-defined, centralised description of the state of the all the organisation’s configuration states. This centralised configuration repository enables the four traditional CM activities identified above and mitigates the risk of loss of configuration knowledge due to the departure of administrative staff.

The rise in popularity of automated configuration management can be attributed to the rise of the related concept of DevOps [4]. In essence, DevOps describes a workflow where operations activities are automated and integrated with development and quality assurance processes. A central tenet is continuous deployment, where new software builds can be deployed to testing, staging or production servers automatically [5]. As a result, configuration becomes another centralised description of the state of the all the organisation’s configuration states. This centralised configuration repository enables the four traditional CM activities identified above and mitigates the risk of loss of configuration knowledge due to the departure of administrative staff.

Simplistic approaches to configuration-as-code, such as shell scripting, have largely been superseded by more modern declarative systems. The declarative approach allows for more conciseness and portability compared to imperative native scripting. The three most popular open source automated software CM tools all follow the declarative approach: Puppet, Chef and CFEngine. The scripts are developed in a collaborative manner on social coding sites and are shared through publicly accessible catalogues or community forums. A brief overview of each of these tools is provided in the following subsections.

A number of commercial systems, such as IBM Tivoli, BMC Bladelogic Automation Suite, HP OpenView and Microsoft Center Configuration Manager are also available. We will not consider them here because development of scripts for these systems is typically performed in-house rather than using collaborative development sites. Access to script catalogues for these systems, if they exist, is typically restricted to paying clients. Ad-hoc sharing of scripts can also occur on forums and mailing lists.

Increasingly, CM tools are being integrated with autodeployment systems. Autodeployment (also known as service orchestration) systems automate the tasks of provisioning the collection of IaaS resources needed for a service and configuring them as needed to bring the service online. A number of autodeployment systems are available, with differing support for multiple service providers, languages and portability.

A. CFEngine

The CFEngine [6] configuration management system is the oldest of the three popular open source CM systems, with development of the first version started by Mark Burgess in 1993. CFEngine has a strong theoretical underpinning in the form of promise theory [7]. The underlying principle is that the system promises to try to achieve the state described in the policy [8]. The configuration might say a specific file has to exist in a fixed location with a certain content and file execution properties. If it cannot fulfil the promise (for example if the permissions of the file are incorrect), it will try to fix the problem in a predefined way by executing subroutines or methods to get to the ideal state. Afterwards, it issues a report describing its actions. The ideal or healthy state is the desired state of CFEngine and the system only tries to reach it, and to get there it will work on one promise at a time. Hence, CFEngine configuration scripts are referred to as bundles with individual clauses referred to as promises.

Another property of promise theory is that every system is only responsible for itself. Promises are not made about the behaviour of other systems. It is possible to create a hierarchy within CFEngine, but this is merely a technique for reducing the load on the main policy hub. The policy hub keeps the configuration files which can be downloaded by the clients.

For security reasons, it is not possible to remotely execute configuration actions directly on client systems. Instead, a remote agent is triggered that causes the CFEngine agent to recheck its promises. The promises can have conditions, like the OS, the type, date, time (specific or night or day), that specify when the promise should be executed. For example, it might be the best to run package updates at night when the user is away from the machine or the load is low. Other conditions include limiting file sizes and, most importantly, the state of a promise. Policies are written in CFEngine’s domain-specific language (DSL). The policy syntax is human readable and hides specific implementation details such as string substitution. The CFEngine agent is lightweight and written in ANSI C. It has minimal dependencies and therefore a much lower system footprint than Puppet and Chef, which both depend on the Ruby toolchain.

Our research has not identified any community specifically for CFEngine bundle sharing. Users help each other in an ad hoc fashion on Google Groups, but there is no official catalogue where bundles are shared.
B. Puppet

Puppet [9] is a CM and service deployment system that has been in development since 2005. Puppet configuration scripts are referred to as manifests. An organisation’s manifests are stored on a centrally managed server or Puppet Master. Development is overseen by Puppet Labs, a commercial entity based on the project that also provides paid support. It is implemented using the Ruby programming language, and hence requires an installation of Ruby to be present on machines that it manages. Some platforms, such as Android, are unsupported as the libraries are not compatible. Puppet’s client-server model has been shown to scale well when configured appropriately [10]. Puppet can be used with cloud based deployments as well as with physical machines. In the latter case it can be used in conjunction with another package, Foreman, for installing the host operating system by using pre-seeding to install the necessary packages after automatically connecting to the Puppet Master. For cloud environments Puppet has a suite of command-line tools to start virtual instances and install puppet on them without logging in manually.

Puppet is quite popular as a CM tool and has built up a large community of users. It is used by organizations such as Nokia, Dell and the Wikimedia Foundation. Puppet Labs provide a manifest sharing platform, Puppet Forge [11], that is used by the community to promote re-use and collaborative development of manifests. Analysis of the manifests available on Puppet Forge indicates that the user base is focused primarily on Linux, especially Ubuntu (209) and RHEL (132) based systems. Windows platforms are supported, although the user base seems to be significantly smaller – only 28 manifests were available at the time of writing.

The manifests available on Puppet Forge are open source and address a wide variety of service deployment and administration tasks. The manifests themselves are written either in a Puppet description language (a DSL using simplified Ruby), or directly in Ruby. The manifests can utilize system facts, that are collected by an information gathering tool called Facter. Information gathered by Facter is used to ensure that only the minimal configuration required to bring a system into the desired state is applied. Two web interfaces are available for Puppet. The first, Puppet Dashboard [10], is developed by Puppet Labs and is used as the interface in the commercial version, although a community version with a reduced set of functionality is freely available. The second interface is Foreman [10], which has more detailed views and integrated support for compute resources, such as cloud service providers. Foreman is an open source project that is built and maintained by a community of volunteers with assistance from Red Hat.

C. Chef

Chef [12] is an open source CM tool and framework developed by Opscode, a company founded in 2008. Furthermore, it exposes APIs and libraries that can be used to build custom tools. The configuration scripts are called cookbooks that consist of recipes. Chef is built on a client-server model, where the Chef client connects to the Chef server to pull the cookbooks and to report the current state. The Chef server had previously been written in Ruby but is now implemented in Erlang [13] for improved performance and scalability. The server is also the reference implementation of the Chef API. The client is written in Ruby and therefore depends on the Ruby runtime. Recipes can be implemented either in pure Ruby or a reduced Ruby DSL.

Chef is divided into multiple components. It has a profiling tool, called Ohai, which gathers data about the system, such as CPU load, network configuration and software versions. It can be extended through plugins to gather specific data that cannot be collected with the default capabilities. The interactive debugging console, Shef, can be used to run the recipes and cookbooks locally for testing purposes to explore how they will behave. The multi-purpose command line tool Knife uses the Chef framework to facilitate system automation, deployment and integration. It provides command and control capabilities for managing virtual, physical and cloud resources across multiple operating systems, including Windows, Linux and Unix.

The cookbooks for Chef are shared with other users through the community web page of Opscode [14]. The website allows tagging, but also allows files to be viewed, including their metadata.rb file, where the developer can add the supported operating systems. Cookbooks are hosted on GitHub, as is the whole Chef project with all its components [15].

D. AWS OpsWorks

AWS OpsWorks [16] is an Amazon product that allows resources to be provisioned via the EC2 service and configured using Chef. Chef recipes can be applied at a number of defined lifecycle stages (Setup, Configure, Deploy, Undeploy, Shutdown). OpsWorks largely supercedes the earlier Amazon Elastic Beanstalk and CloudFormation services. Elastic Beanstalk is limited to deploying applications without adjustments to the platform but allows for easy elastic scaling. CloudFormation uses an Amazon-specific scripting language for CM, preventing the use of the wide variety of publicly available CM scripts. OpsWorks allows deployments to be built as stacks composed of the individual components of the deployment, such as web servers and databases. Cookbooks can be retrieved from SVN and GitHub repositories, HTTP addresses and S3 buckets. Recipes can be executed directly in virtual machine instances, eliminating the need for a dedicated Chef server. OpsWorks is a proprietry hosted service and can be used only with the Amazon EC2 cloud service.

E. RightScale

The cloud service provider RightScale has integrated Chef into their cloud management platform [17]. Cookbooks can be stacked to specify the order in which they should be applied to the instances. Furthermore, stacks can be specified for the boot, operational and decommission lifecycle stages. The cloud management interface does not run a Chef server, instead imitating the way the Chef server works. This sometimes requires modifications to the recipes if specific Chef server functions are used. The Chef cookbooks can be pulled from a tarball file, or an SVN or GitHub repository. If the customer is already using a Chef server, RightScale ServerTemplates for Chef can be used to deploy instances through the private or

1Accessed: 2013-06-26
hosted Chef server. When Chef is paired with ServerTemplates, the configurations are cloud-agnostic, enabling on-demand provisioning across public and private cloud data centres and providers including AWS, CloudStack, Eucalyptus, OpenStack, Rackspace, and SoftLayer. RightScale is a propriety hosted service but supports multiple IaaS providers.

F. Juju

Juju [18] is an open-source service configuration and deployment tool developed by Canonical. It is limited to deploying systems running the Ubuntu Linux distribution. It is compatible with Linux containers, physical machines and cloud environments, such as Amazon EC2, Eucalyptus, OpenStack, HP Cloud and Rackspace. Juju service configuration scripts are called charms and are shared through a public catalogue called the charm store. The charms are stored on the Ubuntu Launchpad platform and can be written in any language that can be interpreted by Ubuntu. This makes it possible to use it in combination with Puppet or Chef, as Juju will just call the agent to run a script or to connect to a Puppet Master or Chef server. Currently it is only possible to assign one charm per instance. Instance specification parameters, such as memory, can be used as prerequisites for charms.

G. Autodeployment extensions to CM systems

Puppet and Chef both have utilities that can provision virtual machines and deploy services directly to them: Cloud Provisioner (for Puppet) and Knife (for Chef). By using these tools it is possible to use a private Puppet Master or Chef server installation which is then used to deploy the services. In both cases it is possible to specify the instance type. This approach differs from RightScale and OpsWorks, as they use an existing system and create cloud instances using the capabilities of the CM tool without being constrained to a particular provider.

III. QUALITY ISSUES WITH CM SCRIPTS

CM systems are generally designed to be as platform-agnostic as possible both in terms of the syntax of their configuration scripts and the portability of the client applications that apply them. Nevertheless, execution environments are highly heterogeneous and there is wide variety in the approaches taken to common configuration tasks such as installing packages, creating files and changing their contents, service lifecycle and user management. Also, these mechanisms are not static and may change over time with newer operating system releases; for example consider Ubuntu’s migration to Upstart from the traditional SystemV-style init daemon. Platform-specific details such as file locations and package names can find their way into scripts, leading to execution failures on platforms where these details differ. Furthermore, the syntax of configuration scripts may change as new versions of the CM system are released, leading to incompatibilities. More subtly, if another language is used to provide the runtime environment (such as Puppet and Chef) then changes to the underlying language’s syntax or libraries can also lead to incompatibilities.

The various failure modes outlined above are examined in detail in subsequent subsections. The root cause of all of these issues is that assumptions about the script’s execution environment may prove to be erroneous when the script is executed on another platform. Portability issues can be resolved by testing scripts on a variety of platforms and adding platform-specific branches as necessary. However, script developers often neglect to test their scripts on platforms other than those that they deploy to within their own organizations. In Section IV we propose a quality assurance platform that would automatically signal portability issues to script developers and consumers.

Note that the quality issues below relate to portability only and hence does not represent an exhaustive list. Non-portability related failure modes might include conflicts between scripts due to mutually exclusive requirements, and global namespace collisions.

A. Operating System Heterogeneity

Obvious incompatibilities in areas such as user and package management are to be expected across fundamentally different platforms such as Microsoft Windows, Solaris, Linux, BSD and Android. Even within ecosystems such as Linux, the decentralized nature of development and distribution has led to a proliferation of individual distributions. Although there have always been differences between, for example, the Debian and the Red Hat ecosystems, nowadays there are also differences between their derivatives (such as Ubuntu and CentOS, respectively) as well. As noted above, the functionality of individual distributions, such as the startup mechanism, may change over time.

The most common incompatibility between different Linux ecosystems in terms of CM is the issue of package name differences. For example, the Apache web server is referred to variously as apache2 (Debian, SUSE), httpd (RHEL) and apache (Arch Linux). There are many of these name differences which also affect the corresponding folders in the filesystem. Another difference is the location of configuration files, such as network interfaces or variables for the initialization scripts of daemons. In Debian-based systems the network configuration file is located in /etc/network/interfaces, with all interfaces in one file, whereas in RHEL based systems an individual file for each interface is located under /etc/sysconfig/network-scripts/.

Writing manifests for Windows systems requires additional steps as there is no centralized package repository where the packages can be retrieved from. For Windows systems the MSI package has to be retrieved beforehand and made available via a file server where it can be accessed by the clients. This can be in a network shared directory or on the configuration manifest server. This approach can be used for other operating systems as well in cases where a custom package or a specific version should be installed, ignoring the latest version in the repository.
As configuration management scripts are supposed to be shareable, those differences make it necessary to add complexity as operating system incompatibilities must be considered on a case-by-case basis. The manifest in Listing 1, for example, will work on Debian, Ubuntu and SUSE systems, but on many other systems it will fail due to package naming differences. To workaround this issue it is possible to add OS-specific behaviour. As the CM agent is aware of the environment it runs in, this information can be used in selecting the correct package name and, if the system is not aware of it, the correct package management system also. The manifest in Listing 2 illustrates this approach. Even though it will work on the five aforementioned Linux systems it will fail on others as the package name is linked to the OS and, even though the package name may be identical, the default behaviour is to fail with an error.

B. Hidden Dependencies

Configuration scripts may have hidden dependencies that are not formally specified but must be present before the script will work successfully. A common scenario is for developers of scripts that install content management systems, such as WordPress, to intentionally omit configuration details for the underlying database. Consumers of these scripts are expected to install the database themselves using another configuration script before updating the script for the content management system to point to this database instance. The author of the script just assumes that the dependency is being taken care of by something else. There are valid technical reasons for this approach, as it allows the same script to be used with MySQL, Postgres, or any database of the user’s choice. However, this flexibility comes at the cost of automation and composability.

An example for such case is the example42/wordpress\(^2\) Puppet manifest. It uses a local MySQL database in its default configuration, but does not install it when absent. Instead the manifest will fail.

C. CM Tool Version Incompatibility

As CM tools get updated frequently to add new functionality, it is possible that the manifest includes commands or constructs that are no longer supported in future versions or were not supported in older versions. As it is customary to keep the version of a tool coherent in a deployment, it is possible that a new manifest will fail for those reasons. For example, in Puppet the language construct `unless` was added in version 3.x [19], which renders the manifest unusable on older Puppet versions.

A related issue can arise due to version mismatches between the server and the client component. Due to internal changes in either of the components it is possible that one does not work with the other. This problem can come up if the server and the agents use different operating systems and/or different repositories that keep different versions. In the case of Chef, upgrading the server from version 10 to 11 can result in client incompatibilities [13].

D. Underlying Platform Incompatibility

Some CM tools are implemented as DSLs for, and/or wrappers around, existing runtime environments. For example, both Puppet and Chef are built around the Ruby language runtime. In contrast, CFEngine uses its own built-in DSL and does not have any external dependencies. The use of an external runtime environment can lead to incompatibilities if an untested version is used. For example, an API change may cause CM script code to break.

E. Installation Mechanism

The installation method of the CM software can also have an impact on script execution. It is possible to install packages either using the centralized distribution repository or a local file server (see Section III-A). In some cases it is also possible to use the mechanism provided by the underlying interpreter, like Ruby GEMS. The fourth way is to install the software from source and compile it manually. This requires the dependencies to be installed, but there might be some incompatibilities between the versions. In the case of Puppet it is possible to install all the GEMS that are necessary, but when the Puppet Master gets installed through the package management system, it does not pick them up. In that case it is necessary to install the corresponding packages to finish the installation. This does not have to be a general problem with a software package, but can result from the way the package was created by the maintainer.

IV. QUALITY ASSURANCE SERVICE

In this section we provide a technical sketch for a quality assurance service for open source CM scripts. The general approach is based on the standard extreme programming practice of continuous integration [20], where a battery of tests is run automatically at defined intervals or events (such as a new code revision being checked in). The idea of applying continuous integration to CM scripts is not new. For example, Puppet manifests developed on GitHub can integrate with Travis CI (see Figure 1), a third-party continuous integration service for the open source community. New builds are, if the source tree is configured correctly, automatically tested using a variety of tools that perform static analysis (using `puppet-lint`) and functional testing across a variety of Ruby and Puppet versions. The results can be integrated into the Puppet Forge manifest catalogue. However, the Travis CI approach addresses only some of the failure modes identified in Sections III: CM tool version incompatibility and underlying platform incompatibility. Operating system heterogeneity is not addressed.

A. Motivation

The primary design goal of our quality assurance service is to independently determine the portability status of CM scripts by automatically applying them across a variety of operating system releases. Additional design objectives include:

1) Support for the three leading CM systems: Puppet, Chef and CFEngine.
2) Static analysis for obvious errors in code by leveraging existing tools: puppet-lint, foodcritic and cflint for Puppet, Chef and CFEngine, respectively.
3) Analysis of compatibility across the range of CM tool versions, e.g., Chef versions 10 and 11.
4) Where possible test alternate installation mechanisms, such as apt-get and Ruby Gems.
5) A publishing mechanism that allows results to be surfaced on third-party websites, such as collaborative development platforms and CM script catalogues.

If all of these objectives were to be implemented then the majority of the failure modes identified in Section III would be addressed.

B. Use case scenario

CM scripts are typically developed with a specific target environment in mind. Tests are run in that environment in order to detect any issues. When all the features are implemented that are deemed important, and it executes without any errors, it is considered to be finished. It is unknown if the script is portable or works only in this environment. This might not be important at the time the script is developed, but incompatibilities may arise when the system gets updated or upgraded, even when the OS or the package selection remain roughly the same. In this case, the developer would benefit from the proposed service, as the testing system would report code quality issues and regressions before problems are experienced. These warnings would allow the developer to improve the quality of the script and address issues before they arise.

From the consumers’ perspective the benefits for the proposed system are more evident. Compatibility issues can be identified during script selection instead of manual testing. This allows problematic scripts to be excluded early in the selection process. It is still necessary to test the script and to check for any parameters that have to be passed along, but this is recommended in any case. The service would allow the consumer to see if the script is compatible with the target OS or not, instead of relying on the documentation and the developers specific testing procedures. By default Linux scripts are tested against Debian, Ubuntu and CentOS, allowing a large user base to be catered for automatically.

C. Technical Sketch of Implementation

For the purpose of our technical sketch we propose to use the Jenkins continuous integration platform in combination with JClouds, but a similar system could be built using Travis CI and Ruby Fog. Both provide a wealth of existing functionality for detecting new source code revisions, executing CI tasks and publishing results. Jenkins was chosen over Hudson as it appears to have greater momentum in terms of development effort (e.g., 431 project members and 890 public repositories for Jenkins on January 22 2013 compared to 33 and 82 for Hudson respectively on the same date). Jenkins integrates well with the Maven build system, producing a .hpi file that can be easily deployed to Jenkins installations.

The JClouds library allows testing to be performed across a variety of operating systems in order to address the issue of operating system heterogeneity. JClouds provides a provider-agnostic interface for launching virtual machines running specified operating system families and versions (see Listing 3). By iterating over a number of operating system families and versions, it is possible to identify those with which problems arise when executing an CM script. The JClouds ComputeService API abstracts the underlying cloud provider. Either a local cloud deployment or a commercial provider could be used during the prototyping process.

The proposed system architecture (see Figure 2) consists of the following: a local Jenkins installation, including the Git and GitHub plugins, is used to monitor the state of a specified GitHub project for a CM script. Whenever a new revision is available, the project is automatically checked out
and our custom Jenkins plugin is executed. The plugin executes a lint tool, such as puppet-lint for Puppet, foodcritic for Chef or cflint for CFEngine scripts, and, if this step succeeds, proceeds to test OS compatibility. OS compatibility is checked by starting a number of virtual machine instances on the local cloud deployment using the JClouds API. The JClouds runScriptOnNode API call is used to install the respective client or agent, check out the manifest being tested, and retrieve the results. Existing Jenkins publishing methods are used to surface the output of each run. The plugin needs to be written in a modular fashion so that additional CM systems and testing modes (such as CM tool versioning) can be added easily.

![Diagram of QA architecture for CM scripts](image)

**Fig. 2.** QA architecture for CM scripts. This approach is exemplified by Travis CI, bar cloud support.

The existing sharing platforms for CM scripts, such as Puppet Forge and Opscode Community, provide functionality beyond simply listing scripts. They support script descriptions and tagging to help consumers to find appropriate scripts. Additional information, such as the number of downloads and time of last update, assist with this process. The Chef community can furthermore leave comments on cookbooks and mark them. This helps to build trust, but is inconsistent and relies on human intervention. The addition of Travis CI badges adds an objective validation and surfaces compatibility. The external and live character of this badge emphasise the value, as the testing service is not related to the main project. Further trust is generated by open development, where everybody can access and review the source code and raise issues on the development platform, in a transparent fashion.

Our proposed QA service would allow portability and other quality information to be surfaced in a similar fashion. This would add benefits for the developers, as it shows the compatibility and the results of the static analysis directly, which might engage other developers to add missing compatibilities or change the code to match other operating systems. Meanwhile consumers can immediately identify CM scripts that will work in their environment without having to rely on the completeness of the script tags.

**V. SUPPORT FOR AUTODEPLOYMENT**

CM scripts that are packaged as part of larger autodeployment specifications can be expected to operate correctly. This is because their environmental parameters, such as operating system and CM tool versions, are specified as part of the larger deployment and the entire assembly is tested in unison during development. However, autodeployment introduces another failure mode: incompatibilities between cloud providers and even between different regions of the same cloud provider. Differences between cloud providers are to be expected, as many implement common APIs, such as the EC2 web service API, to differing extents. Incompatibilities between provider regions are more surprising but do occur. For example, in the authors’ experience some Juju deployments were found not to work correctly on EC2 regions other than US-East.

A possible future expansion of our proposed CM testing platform is therefore the ability to run autodeployment scripts across multiple cloud service providers and regions to expose incompatibilities. As a result, consumers of autodeployment scripts could be assured that an autodeployment script is likely to work with their chosen provider or region. The provider-agnostic nature of the JClouds API would allow the required functionality to be implemented by extending the existing code-base.

**VI. RELATED WORK**

Pham et al. [21] [22] present an analysis of the testing culture on social coding sites. GitHub and other social coding sites are becoming more popular as they lower the barriers to entry for engagement in software projects. New developers can obtain and contribute to code easily. Much of this functionality is built on the availability of decentralized software version control systems such as Git. Integration of version control with continuous integration allows the testing status of a project to be recomputed after each change. Continuous integration results can be communicated as a badge that can be integrated into the project web page on the social coding site to generate trust and assurance.

Pandey [23] has investigated the communities of Chef, Puppet and CFEngine. Areas of consideration include community size, their contributions and activities, bug reporting and bug fixing. A combination of mailing-list and search results was used as proxies for measuring general interest in each tool. Furthermore a usability test sequence was performed to identify which tool is most appropriate for a particular task.

Delaet et al. [24] present a comparison framework for CM tools. The framework allows system administrators to compare the systems using a number of factors and hence identify the one that most suits their particular use case. The framework covers not only open source tools, but also proprietary tools, such as IBM Tivoli or Microsoft Server Center Configuration Manager (SCCM).

Security concerns around software CM have been identified [25], [26]. Accidental or malicious disclosure of sensitive CM scripts could reveal details of internal processes and IP. Another security risk comes from malicious CM scripts, where an attacker releases a script that creates security holes, intentionally interferes with other processes, or installs and distributes malware.

Lynn et al. [27] propose active and dynamic trustmarks for building trust in cloud services. A key insight is that trustmarks need not be static badge-like entities. Instead, they can be live
animated entities that provide a graphical representation of underlying quality metrics. In this case, the underlying metric may be an automated compatibility test results.

VII. CONCLUSION

Automated CM confers many benefits on organizations that adopt it. Forums for sharing configuration scripts have emerged, encouraging re-use. We have identified a number of quality issues that can arise when scripts are shared without being tested extensively across the target platforms that they are likely to be executed on. Furthermore, we demonstrated that a platform-agnostic system of automated testing could be used to address these quality issues by surfacing incompatibility information to script developers and consumers. Additional failure modes arise when the practice of automated configuration is combined with that of automated deployment via IaaS. We described how the automatic testing platform could be extended to test for these.

VIII. ACKNOWLEDGEMENT

The research work described in this paper was supported by the Irish Centre for Cloud Computing and Commerce, an Irish national Technology Centre funded by Enterprise Ireland and the Irish Industrial Development Authority.

REFERENCES