Automated Cross-Platform Compatibility Testing for Infrastructure as Code

MATTEO NUSSBAUMER[*] and EIMAN ALNUAIMI[*], ETH Zurich, Switzerland

As cloud computing and DevOps practices continue to evolve, Infrastructure as Code (IaC) has become a crucial methodology for configuring and managing infrastructure through executable source files. Despite its benefits, IaC lacks robust automated testing frameworks tailored to its unique characteristics, leading to potential system crashes and security vulnerabilities. This paper presents an automated testing framework designed to verify the cross-platform compatibility of Ansible automation scripts. Our contribution is a modular testing framework that can be extended to accommodate any number of Ansible playbooks, test oracles, and operating system containers. Utilizing black-box functional testing and custom test oracles, our framework validates the functional requirements of Infrastructure as Code (IaC) deployments across various operating systems, including Debian and Red Hat-based distributions. Experimental results demonstrate that our domain-specific test oracles operate effectively on Ansible playbooks with similar functions. The framework's modular architecture and significant speed make it scalable while maintaining good performance. This work represents a significant step towards ensuring reliable and robust IaC deployments across diverse environments.

1 INTRODUCTION

As cloud computing and DevOps practices continue to rapidly expand, Infrastructure as Code (IaC) has emerged as a fundamental approach to configure and manage infrastructure. Instead of manual configuration, executable source files are employed to facilitate reusability and automation [11]. Unlike traditional software applications, which benefit from established testing methodologies and practices, IaC configurations often lack robust automated testing paradigms tailored to their unique characteristics. The absence of systematic testing approaches for IaC leads to many potential pitfalls including system crashes and security vulnerabilities and leaves the infrastructures vulnerable to unforeseen errors and inconsistencies[6, 13].

The need for automated testing is further emphasized by the diverse landscape of operating systems and versions upon which these infrastructures may be deployed. There are nearly 250 different active Linux distributions today, of which 54 are server distributions [1, 2]. These unique deployment environments make it challenging to ensure reusability and reliability. Additionally, neglecting cross-platform compatibility testing can result in significant limitations on the user base, as different users may require different operating systems or versions for their deployments. Therefore, automated cross-platform testing plays a crucial role in mitigating risks associated with platform discrepancies and expanding the reach of IaC solutions without compromising their functionality or stability.

In this paper, we propose a novel solution that automates testing of Ansible automation scripts across diverse operating systems. We apply black-box functional testing to the deployed infrastructure to verify behaviour correctness based on specifications and consistent functionality using custom test oracles, which include both domain-agnostic and domain-specific oracles. Additionally, we investigate whether the domain-specific test oracles can be extended to Ansible's modules which offer similar functionality as our deployed infrastructure. The aim of this paper is to provide an automated testing tool for Ansible's playbooks under different configurations, while also exploring

[*]Both authors contributed equally to this research.

Authors' address: Matteo Nussbaumer, matteon@student.ethz.ch; Eiman Alnuaimi, ealnuaimi@student.ethz.ch, ETH Zurich, Zurich, Switzerland.

the limitations of using Ansible on various operating systems, assessing its interoperability, and searching for potential bugs that may arise.

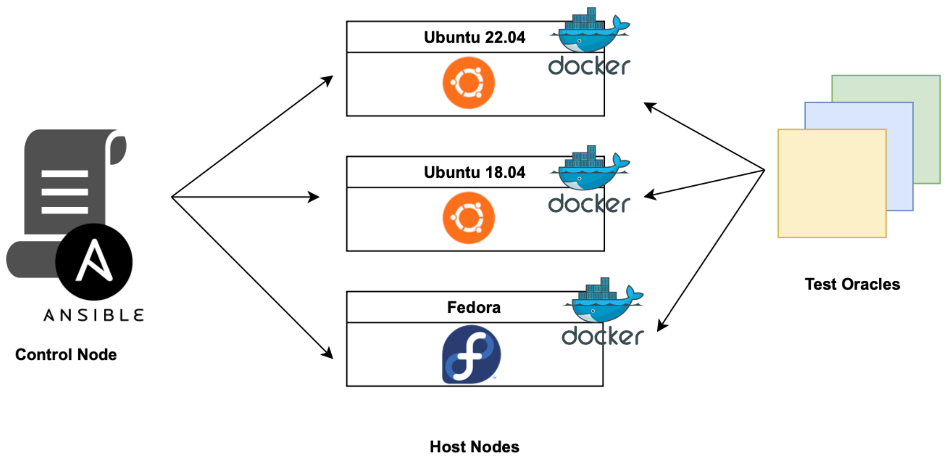

Fig. 1. The Proposed Automated Testing Approach for Ansible Playbooks Across Different Platforms

2 RELATED WORK

In recent years, significant endeavors have been made to devise frameworks for testing infrastructure as code. A model-based testing framework has been proposed to verify the convergence of Chef's automation scripts known as recipes to a desired state through a series of idempotent steps [7, 8]. Another example of an IaC testing framework is ProTI, an automated unit-testing tool for programming languages IaC (PL-IaC) such as Pulumi Typescript. As a first use-case the authors implemented a ProTI-plugin for property-based testing[17, 18]. It makes use of the type property of input and output configurations in Pulumi to generate meaningful values for the variables in the automation script, thus test cases to find bugs reliably. Rehearsal is a sound and complete static verification tool designed for Puppet configurations, also known as manifests, that find non-deterministic bugs and enforce idempotency through determinacy analysis [15]. Moreover, Sharma et. al [16] employ a code-smell detection approach to identify and categorize bugs in Puppet manifests. They utilize an existing code-style checker tool called Puppet-Lint [12] to detect implementation configuration smells and introduce their own tool, Puppeteer, to identify design configuration smells. Rahman et al. [14] detect defects in Puppet manifests using an in-house prediction model trained on text features extracted from open-source repositories.

All these testing frameworks require access to the source code. In the first example, IaC scripts are used as input to derive a state transition graph (STG) that is used to test idempotence of scripts. Similarly, Rehearsal compiles the manifests into a resource graph. It models the semantics of each resource using small programs written in FS, a low-level imperative language for file system operations. Then, it applies the proposed determinacy analysis, producing logical formulas that are later fed to an SMT solver. Moreover, ProTI requires instrumentation of the automation script before employing their proposed type-based test generators and oracles. Sharma et al.'s work requires the smell-detection tools to have direct access to the configuration source code. In the same way, Rahman et al. require access to various configuration scripts to train their model, and the prediction is run on configuration source code as well. Unlike these testing frameworks, our work proposes a blackbox testing approach that doesn't require knowledge of the automation scripts source code.

To the best of our knowledge, we are the first work that considers cross-platform compatibility testing for IaC despite it being a common testing methodology for applications development. [9] tackles the issue of testing the user interface (UI) of cross-platform mobile apps. As each platform has a specific way of representing UI elements, various test scripts are required to test hybrid-apps, one script for each platform it supports. This work proposes 8 different test methods and consolidates them in one single tool to reduce the number of platform-specific test scripts. [10] is another example that presents an automated cross-browser testing method to validate functionality of web apps on the client side using behavioural coverage.

3 APPROACH

3.1 Overview

To test Ansible automation scripts across different platforms, we implement a testing framework which is sketched in figure 1. An Ansible control node orchestrates the infrastructure deployment across multiple host devices that run a spectrum of operating systems. As an example, we designate the Apache HTTP server as our chosen deployment infrastructure. We examine various operating systems and distributions. To emulate real-world cloud deployments while minimizing resource consumption, we employ Docker containers as host nodes. However, our framework can be also be used with different kind of hosts such as virtual machines or servers running an operating system naively.

To ensure Ansible's configuration scripts function as intended on various operating systems, we validate the necessary functional requirements expected from a server by employing various test oracles, some are domain-specific and others are not, each addressing a distinct functional aspect of the infrastructure. This approach will help us find platform specific bugs and study the interoperability of Ansible modules across different platforms. We further investigate the efficacy of the domain-specific oracles on other Ansible modules that offer the same functionality as the Apache HTTP server.

3.2 Operating Systems Selection and Docker Containers

Considering different operating systems is essential because each has its own command-line interface and uses different commands for tasks such as installing software, configuring services, and managing files. This variability makes Ansible's automation scripts challenging to use and test across different platforms. To create a realistic testing environment, we focus on the popularity and availability of platforms. In this paper, we concentrate on various Linux distributions before extending our framework to other operating systems. Linux distributions offer diverse testing scenarios, covering a range of configurations and use cases. Our proof of concept focuses on Debian-based distributions, and we later test how well our framework can be extended to Red Hat-based distributions, identifying necessary modifications. The operating systems we select are Ubuntu (18.04 and 22.04) and Fedora 39.

We utilize Docker containers to host various operating systems, leveraging their lightweight nature and ability to run in parallel, which makes them ideal for our automated testing framework. Docker enables the creation of containers from images, starting from a lightweight base image. As users make changes to the container, a data layer is added on top of the base image. The base images we are using are not pre-loaded with software packages. We hypothesize that this lack of pre-installed services will complicate the setup process for each container, making it a unique task for each specific operating system.

3.3 Ansible's Configuration Automation

Ansible uses an inventory file and a playbook written in YAML on a control node to set up a target infrastructure and manage the configuration of remote hosts. The inventory file lists the hosts Ansible will manage, along with their connection details. The playbook defines the tasks to be executed on these hosts using Ansible modules. When Ansible is executed on the control node, it connects to each host listed in the inventory file using SSH and executes the tasks defined in the playbook on these hosts. Ansible playbooks consist of one or more plays. We will focus on testing one play at a time to achieve unit testing.

3.4 Verification of Functional Requirements

We employ black-box functional testing to create our test oracles. This approach makes it difficult to develop meaningful test oracles without the internal workings of Ansible automation scripts. However, we focus on the observable behavior of the automation script and the target infrastructure. We define one domain-agnostic test oracle applicable to all modules, irrespective of their functionality or target configuration. The remaining oracles are domain-specific and may require adjustment by users based on their target infrastructure and configuration. We assess the extent to which these oracles can be used without modification and determine the level of adjustment needed for Ansible modules with similar functionality, such as Nginx. Our validation framework can be extended with more domain-specific and generic oracles but for now we define the following test oracles:

1. *Test Oracle One:* The first test oracle is applicable to all Ansible modules regardless of their functionality or target configuration. The functional requirement it proves is all tasks within the playbook should execute without generating any errors. It relies on Ansible's output, specifically the "Play Recap" section, which provides a summary for each targeted host during playbook execution. This oracle verifies that the number of failed tasks in a given playbook is zero. Three parameters are relevant for this task: "failed," indicating the number of tasks that failed to execute on the host; "unreachable," representing the number of hosts that Ansible couldn't reach due to network issues or incorrect SSH settings; and "ignored," indicating the number of tasks that executed with errors but had *ignore_errors: true*. This setting allows Ansible to continue executing tasks on that host despite the failure, as by default, it stops executing on a host if it encounters a failure.
2. *Test Oracle Two:* This oracle tests the functional requirement that the server must be accessible and responsive to incoming requests. It achieves this by sending a HEAD request using the cURL command. HEAD requests is a common way to check the status of a server without downloading the entire content. The test expects the server to respond with a 200 OK status code, indicating successful access. Certain status codes, such as 404 Not Found or 503 Service Unavailable, indicate that the server is unreachable.
3. *Test Oracle Three:* The functional requirement tested by this oracle is the implementation of security measures to minimize the attack surface and mitigate the risk of unauthorized access. The oracle utilizes nmap to scan for open ports and performs TCP/IP fingerprinting to identify potential vulnerabilities.

4 IMPLEMENTATION

4.1 Modular Architecture for Diverse Host Environments

To ensure that our testing framework can be used with different host environments (i.e. Docker containers, VMs, native OS servers) and testing oracles, we designed it to be modular. The framework's core component is the test runner, responsible for executing playbooks on a specified list of hosts, gathering data, and running the oracle for each deployment. Therefore we define an interface for the host generator that provides two methods using Python's Abstract Base Classes (ABC). A *get_generator* method that returns an iterable generator returning an object that contains all the necessary information to deploy a playbook on the generated host. A *destroy_host* method that performs actions to tear down the generated host if necessary. The runner gets an implementation of a host generator which for each run sets up a host and destroys it after that run. The functionality of the test runner is visualized in figure 2. For each generated host, the generator returns an inventory file that the test runner can use to deploy the playbook and run the tests. In this project, we implement a host generator creating Docker containers for different operating systems, that are destroyed after each run. Additionally, we define an interface for Oracles that provides the method *verify_deployment* which takes the Ansible logs of a deployment and the Host object generated by the host generator. The method returns a TestResult object that contains information about whether the deployment was successful or not and error messages. The oracle can then analyze the Ansible logs and communicate with the host to verify if the deployment was successful. A test with our framework is sketched with pseudo-code in figure 3.

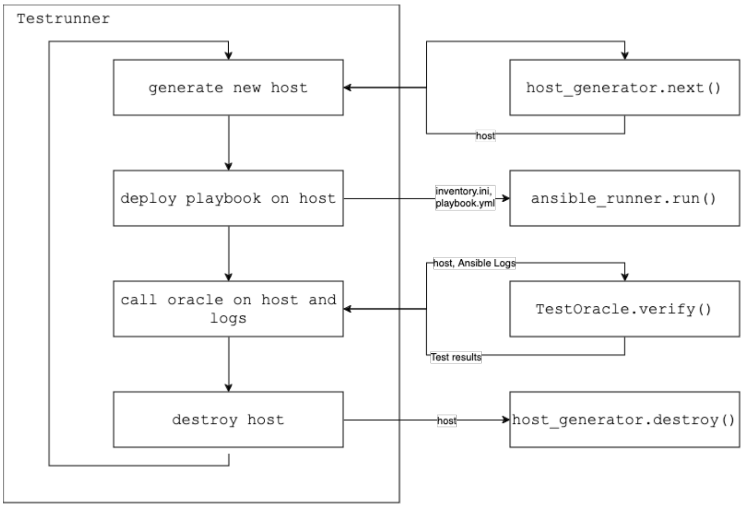

Fig. 2. Sketch of the TestRunner

4.2 Test Oracles Implementation

Each test oracle is implemented as a class that inherits from the `TestOracle` class. This approach allows for easy expansion of the framework to accommodate different deployment infrastructures and facilitates the flexible addition of more oracles as needed. As explained in Section 4.1, each test oracle class contains a `verify_deployment` method that implements the tests for the infrastructure. To maintain a black-box testing approach, the commands in the tests are not executed inside the containers but instead inspect Ansible playbook recaps and use network testing tools like cURL and nmap to verify the deployment of the HTTP server. In figure 4, we illustrate the implementation of the aliveness oracle in our testing framework.

4.3 Ansible Playbook for Apache HTTP Server

Different operating systems use various package managers and modules to set up services and software. For example, Debian-based Linux distributions use *apt*, while Red Hat-based distributions

```python
# Instantiate HostGenerator that generates Docker hosts
# with 4 different operating systems
host_generator = DockerHostGenerator(['ubuntu_18', 'ubuntu22', 'fedora', 'centos'])
# Instantiate SimpleDeploymentOracle that checks if the deployment was successful
# by reading the Ansible logs
test_oracle = SimpleDeploymentOracle()
# Instantiate TestRunner that runs the deployment test
test_runner = TestRunner(host_generator = host_generator,
                         playbook_path = 'deploy.yml',
                         test_oracle = test_oracle)
# Run the test
test_runner.run()
```

Fig. 3. Example of a test using the test runner

```python
class AlivenessOracle(TestOracle):
    def verify_deployment(self, host: Host, deployment_data: DeploymentData) -> TestResult:

        # Construct the curl command to send a HEAD request to check server status
        curl_command = f"curl -I http://{host.ip_addr}:8080 -o /dev/null -w '%{{http_code}}' -s"

        try:
            # Execute the curl command
            http_status = subprocess.run(
                curl_command, shell=True, check=True, text=True, capture_output=True
            ).stdout.strip()
            # Check if the HTTP status code is '200', indicating success
            if http_status == "200":
                return TestResult(True, "Deployment verified successfully")
            else:
                return TestResult(False, f"Server response with status code: {http_status}")

        except subprocess.CalledProcessError as e:
            # Handle cases where the curl command fails
            return TestResult(False, f"Failed to connect to the server: {e}")
```

Fig. 4. Implementation of the Aliveness Oracle in the testing framework

use *yum* or *dnf* to install packages. To minimize manual configuration while using our testing framework, we leverage *ansible_facts*. By default, at the start of playbook execution, Ansible identifies the type of OS using its built-in *ansible_facts*. The *ansible_os_family* fact is particularly useful for determining the OS family (e.g., Debian, RedHat, etc.). We reference the *ansible_os_family* fact in our playbook to conditionally execute tasks based on the detected environment.

Figure 5 shows the Debian-based part of the Ansible playbook that automates the deployment of the Apache HTTP server on Ansible hosts. It begins by installing the Apache2 package using the *apt* module, ensuring the latest version is retrieved by updating the package cache. Next, it ensures the Apache service is both started and enabled to automatically start on boot using the *service* module. Finally, it configures the firewall to allow HTTP traffic on port 80 by enabling the "Apache Full" profile through the *ansible.builtin.ufw* module. This playbook effectively handles the installation, configuration, and firewall setup required to deploy a functional Apache HTTP server.

4.4 Technical Implementation Details

Our framework is implemented with Python 3.12.3 [4].For the Docker host generator we use the Docker SDK for python [3] to pull images for different operating systems and deploying them with a local Docker engine instance. The test runner uses the python package *ansible-runner* to deploy Ansible playbooks [5] and read out the logs.

```yaml
---
- name: Deploy Apache HTTP Server on Ubuntu/Debian
  hosts: all
  become: yes

  tasks:
    - name: Ensure the apt package list is updated (Debian-based systems)
      apt:
        update_cache: yes
      when: ansible_os_family == "Debian"

    - name: Install Apache2 on Debian-based systems
      apt:
        name: apache2
        state: present
      when: ansible_os_family == "Debian"

    - name: Ensure Apache is running and enabled on Debian-based systems
      service:
        name: apache2
        state: started
        enabled: yes
      when: ansible_os_family == "Debian"

    - name: Allow Apache through UFW on Debian-based systems
      ufw:
        rule: allow
        name: "Apache Full"
      when: ansible_os_family == "Debian"
```

Fig. 5. Ansible Playbook to Deploy Apache HTTP Server

Since the base image of the Docker containers lacks necessary tools and packages, they must be set up with essential utilities such as Python 3, sudo, and ufw (Uncomplicated Firewall). Python 3 is crucial for Ansible host nodes because Ansible's modules and execution environment rely on Python. It serves as the underlying language for many Ansible modules, and its presence is necessary for running playbooks and modules on managed hosts. The firewall is essential to allow TCP traffic into the container and to enhance security by managing firewall rules.

Moreover, the containers need to be configured for network communication so that test oracles can reach them and validate the deployed infrastructure. This is achieved by creating a custom Docker network with a specified subnet. Upon launch, each container is assigned an IP address within this network, created using the bridge driver. The bridge network driver provides isolation between containers by creating a virtualized network interface for each container and placing them in separate network namespaces. Containers connected to the bridge network can communicate with the host machine through the host's IP address and exposed ports. To test the aliveness of the HTTP infrastructure, TCP port 80 of the container is mapped to port 8080 on the host machine. Consequently, network traffic directed to port 8080 on the host is forwarded to port 80 on the container, enabling seamless testing and interaction with the deployed HTTP service.

5 EXPERIMENT AND RESULTS

5.1 Experiment Settings and Environment

The project development and evaluation is done Apple MacBooks equipped with Apple Silicon M1 processors, 32GB RAM and MacOS Sonoma operating system. The software versions that were used are:

- python 3.12.3
- ansible 9.5.1

- ansible-runner 2.3.6
- docker 7.0.0.

During the development of our testing framework, we concentrated on Debian-based distributions and aim to evaluate how well our framework can be extended to Red Hat-based distributions. The docker images that were used to deploy and test the framework are:

- Ubuntu 22.04
- Ubuntu 18.04
- Fedora 39

5.2 Functional Validation of Apache HTTP Server Deployment

We have tested our automated framework by deploying the Apache HTTP server across multiple Docker containers running different operating systems, including Ubuntu 18.04, Ubuntu 22.04 and Fedora 39. The deployment was orchestrated using an Ansible control node, which executed the playbook designed to install, configure, and start the Apache server on each host.

During the testing phase, the test oracles, defined in section 3.4, were applied to verify the functional requirements of the deployed infrastructure. Table 1 summarizes which test oracles passed or failed to verify the deployment of the Apache server across various Linux distributions. The deployment of the Apache HTTP server on Debian-based distributions passed all tests, and the test oracles worked as expected. We explain why the test oracles failed on Red-hat based distributions in section 5.4.

5.3 Testing Flexibility with Nginx Server Deployment

To evaluate the flexibility and applicability of our testing framework, we conducted tests using a different Ansible playbook that deploys an Nginx server instead of the Apache HTTP server. This experiment aimed to assess whether the domain-specific test oracles could be effectively reused for Ansible modules with similar functionality and to verify the consistency of our testing approach across different deployment scenarios. The Nginx playbook was designed to install, configure, and start the Nginx server on the same set of Docker containers running the same operating systems from the previous experiment.

As shown in Table 1, the deployment of the Nginx server passed all tests, and the test oracles worked as effectively as they did for the Apache HTTP server on the Debian-based distributions. This outcome highlights the versatility and robustness of our testing framework, demonstrating its capability to handle different types of infrastructure as code deployments. The successful reuse of the test oracles for Nginx further underscores the framework's potential for broader application across various Ansible modules, ensuring functional and security compliance in diverse deployment environments.

	Apache			**Nginx**		
	Test1	**Test2**	**Test3**	**Test1**	**Test2**	**Test3**
Ubuntu 18.04	✓	✓	✓	✓	✓	✓
Ubuntu 20.04	✓	✓	✓	✓	✓	✓
Fedora 39	x	x	✓	x	x	✓

Table 1. Deployment verification results for Apache and Nginx servers using test oracles across different Linux distributions

5.4 Cross-Platform Compatibility Testing on Different Linux Distributions

To evaluate the interoperability and robustness of our testing framework across different operating systems, we deployed and tested the Ansible playbook on Red Hat-based Linux distributions, specifically Fedora, alongside the previously tested Debian-based distributions. This assessment aimed to identify the necessary adjustments and configurations required for seamless cross-platform compatibility.

During our tests, we encountered several challenges related to package availability and system service management. For example, the Ansible playbook typically uses the *service* command to start the HTTP server on Debian-based images, where this command is available by default. However, in Red Hat-based images like Fedora, the *service* module is not included by default, necessitating the explicit setup of *systemd* to use *systemctl* or the *service* command. This discrepancy required us to build custom images rather than relying solely on pre-built ones.

Additionally, our framework initially treated all containers uniformly, which proved inadequate for platforms with different entry point requirements. For instance, while Debian-based containers could use "/bin/bash" as the entry point, Fedora containers required the *systemd* service script as the entry point. Addressing these nuances was crucial for accurately deploying and managing containers across various operating systems.

Our experiments confirmed that the framework could be extended to support different Linux distributions by incorporating these specific adjustments. The successful deployment and testing of the Ansible playbook on Fedora highlighted the need for conditional configurations and the flexibility to build custom images tailored to the target operating systems.

5.5 Performance and Scalability Prospects

To assess whether our testing method can be applied in real-world scenarios we estimate the scalability of our framework. For this purpose we measured the execution times of the three basic steps of our framework (Table 2):

- **Setup time:** The duration required for setting up the host (in this case a docker container)
- **Deployment time:** The duration of playbook deployment
- **Verification time:** The duration of the deployment verification process performed by our three oracles

Upon reviewing the measurements, it becomes evident that setup and deployment clearly dominate the overall execution time. This means that our framework puts only a little constant overhead on top of the deployment time of the playbook under test. For our relatively simple playbooks the total duration of the test is on average at most 32.21 seconds per host, which seems reasonably fast compared to execution times of normal sized unit and integration test suite. Unfortunately we were unable to find any reliable references for comparison. Running the testing framework on all containers simultaneously could further speed up our testing framework since each of the main testing steps can be executed independently for each host. This could mean a speedup of almost three times compared to our current implementation. These measurements prove that our proposed automated testing framework for Ansible playbooks can be scaled while maintaining significant speed.

6 FUTURE WORK

Currently, our testing framework successfully deploys and runs tests on the target infrastructure for Debian-based Linux operating systems only. While our Ansible playbook is tailored to support both Debian and Red Hat-based Linux operating systems, and we do support a Fedora container,

	Setup time	Deployment time	Verification time	Total exec. time
Nginx	15.52	10.86	3.15	29.53
Apache	15.55	13.51	3.16	32.21

Table 2. Execution times (in seconds) measured for the Nginx and Apache playbook separately and averaged over 10 runs and the three tested hosts.

our framework needs further development to support a broader range of Linux distributions and other operating systems.

In future work, we aim to extend the framework's compatibility to support building custom Docker images, which will include the necessary services and configurations for different operating systems. Additionally, we will refine the framework to handle different entry points and initialization scripts for various containers, ensuring accurate and reliable deployments. These enhancements will enable our testing framework to be more versatile and robust, capable of supporting a wider range of environments and operating systems.

7 CONCLUSION

In this report, we introduced a novel automated testing framework designed to verify the cross-platform compatibility of IaC deployments, with a focus on Ansible automation scripts. By leveraging black-box functional testing and custom test oracles, our framework successfully validated the functional and security requirements of Apache and Nginx server deployments across multiple operating systems, including various Debian and Red Hat-based distributions.

Our experiments demonstrated the framework's robustness and versatility, effectively identifying and addressing platform-specific issues and ensuring consistent functionality across different environments. The successful reuse of domain-specific test oracles for different Ansible modules further underscored the framework's flexibility.

While the current implementation supports Debian-based systems and Fedora, future enhancements will focus on supporting a broader range of Linux distributions and other operating systems by building custom Docker images and refining entry point configurations. These improvements will enable more comprehensive cross-platform compatibility testing, ensuring the reliability and stability of IaC deployments in diverse environments.

Overall, our automated testing framework represents a significant step towards mitigating risks associated with platform discrepancies in IaC deployments, offering a scalable and adaptable solution for maintaining the integrity and performance of infrastructure across varied operating systems.

REFERENCES

[1] [n. d.]. *DistroWatch.com: Active Linux Distributions.* https://distrowatch.com/search.php?ostype=Linux&category=All&origin=All&basedon=All&notbasedon=None&desktop=All&architecture=All&package=All&rolling=All&isosize=All&netinstall=All&language=All&defaultinit=All&status=Active#simple

[2] [n. d.]. *DistroWatch.com: Active Linux Server Distributions.* https://distrowatch.com/search.php?ostype=Linux&category=Server&origin=All&basedon=All&notbasedon=None&desktop=All&architecture=All&package=All&rolling=All&isosize=All&netinstall=All&language=All&defaultinit=All&status=Active#simple

[3] [n. d.]. *Docker SDK for Python — Docker SDK for Python 7.0.0 documentation.* https://docker-py.readthedocs.io/en/7.0.0/

[4] [n. d.]. *Python Release Python 3.12.3.* https://www.python.org/downloads/release/python-3123/

[5] [n. d.]. *Using Runner as a Python Module Interface to Ansible — ansible-runner documentation.* https://ansible.readthedocs.io/projects/runner/en/2.3.6/python_interface/

[6] Mohammed Mehedi Hasan, Farzana Ahamed Bhuiyan, and Akond Rahman. [n. d.]. Testing practices for infrastructure as code. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing* (New York, NY, USA, 2020-11-08) *(LANGETI 2020)*. Association for Computing Machinery, 7–12. https://doi.org/10.1145/3416504.3424334

[7] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Automated testing of chef automation scripts. In *Proceedings Demo & Poster Track of ACM/IFIP/USENIX International Middleware Conference.* 1–2.

[8] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Testing idempotence for infrastructure as code. In *Middleware 2013: ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings 14.* Springer, 368–388.

[9] Andre Augusto Menegassi and Andre Takeshi Endo. 2020. Automated tests for cross-platform mobile apps in multiple configurations. *IET software* 14, 1 (2020), 27–38.

[10] Ali Mesbah and Mukul R Prasad. 2011. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering.* 561–570.

[11] Kief Morris. 2016. *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.".

[12] Puppet-lint. [n. d.]. Puppet-lint: Puppet code style checker. Last accessed on: 25th April 2024. http://puppet-lint.com

[13] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. [n. d.]. A systematic mapping study of infrastructure as code research. 108 ([n. d.]), 65–77. https://doi.org/10.1016/j.infsof.2018.12.004

[14] Akond Rahman and Laurie Williams. 2018. Characterizing defective configuration scripts used for continuous deployment. In *2018 IEEE 11th International conference on software testing, verification and validation (ICST).* IEEE, 34–45.

[15] Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN conference on programming language design and implementation.* 416–430.

[16] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *Proceedings of the 13th International Conference on Mining Software Repositories.* 189–200.

[17] Daniel Sokolowski and Guido Salvaneschi. 2023. Towards reliable infrastructure as code. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C).* IEEE, 318–321.

[18] David Spielmann, Daniel Sokolowski, and Guido Salvaneschi. 2023. Extensible Testing for Infrastructure as Code. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity.* 58–60.