Automated Testing of Infrastructure as Code (IaC) programs

Théo Ducrey
tducrey@student.ethz.ch

Fedor Miloglyadov
fmiloglya@student.ethz.ch

March 2024

1 Abstract

In this project, we present an approach to practical fault detection in Puppet programs. Our project is programmed in Python and leverages techniques from software testing to identify potential faults and anomalies in Puppet manifests.

We introduce a comprehensive framework that encompasses various stages of fault detection, including processing of trace data, metamorphic testing, state reconciliation analysis, and at the end a summary of the result. Through empirical evaluation on real-world Puppet deployments, we show the effectiveness of our approach in detecting and preventing configuration errors, thus improving the robustness and reliability of infrastructure as code systems, that are Puppet-based.

The project code is accessible here:
Github Link

2 Introduction

With the growing use of cloud computing and the steadily increasing amount of services available for these Infrastructures, the management of these systems becomes intractable to do by hand, especially on larger scales. That's why Infrastructure as Code (IaC) tools and services can automate the creation and maintenance of these systems. These tools, often have many different popular modules, Puppet for example, Puppet has manifests, written for it, and for the Puppet to properly work, each of these manifests has to work properly. To test these modules one possibility is to do static testing, but this, although thorough with known faults, will get intractable at larger scales and will often miss bugs that are in states, that the testers didn't think of. The second option is to run the module on morphed starting states systematically and thus simulate realistic scenarios and their edge cases. This approach, although not as thorough as static testing, since it does shallow testing over a larger amount of start states, allows to cover a larger amount of states, which would be intractable to do by hand with static testing, and this is what we will be doing in this work. For this project, the IaC we will be looking at is Puppet.

One important quality of IaC methods and tools is state reconciliation, this property means that no matter what the starting state is, the end result will always be the same. In the Puppet tool for example, where it uses manifests, that contain actions that are supposed to be taken, for example, it has to install and set up a program. Now the starting state of the system, that is running that manifest can be that it doesn't have that program, that it has the program installed, but not yet set up, or that it already has it set up. In all three cases, the end result has to be the same, that the program is installed on the system and is set up. To do this manifests have dependencies for the actions so that the state reconciliation condition is fulfilled.

For this project, an important testing strategy will be metamorphic testing. The idea of this testing strategy is that we have a base system state on which we run our program on and morphed base system states on which we then run our program as well. Then the results are compared. The morphing can be done randomly and also be done systematically with an algorithm since just random perturbations rarely will lead to the desired edge cases one wants to test for. The goal is to find those states and dependencies that will lead to a different outcome, then run the program on the base system. In our case for testing the manifests, our goal is to make the manifest run incorrectly by first observing it run on previous states and from each run observation create a new state that might lead to an incorrect run.

3 Related work

3.1 Asserting Reliable Convergence for Configuration Management Scripts

This paper is one of the most important for our project. The idea of non-static testing of Infrastructure as Code tools comes from this paper and is also the basis of the second paper which im-

proves and accelerates it. In this paper they use model-based testing by tracing the changes to the state of a Puppet manifest, modeling these changes as a state transition graph and using it to test for idempotence and convergence. This is done in part by defining a property *preservation* and using it to test for these bugs. Tracking the state of the system will be critical to our project, from which we then extract the basic building blocks of the trace. From these basic blocks we then induce which initial state could potentially not lead to state reconciliation. Another large part of this paper, which will lay the groundwork for our project, is the plethora of definitions and logical conclusions about *preservation*, since our approach is to some extent the approach of this paper, where in parallel to checking the trace for preservation, we need to infer which states could lead to a violation of preservation.

3.2 Practical Fault Detection in Puppet Programs / Detecting Missing Dependencies and Notifiers in Puppet Programs

This paper serves as the cornerstone for our research, providing the framework upon which our work is constructed. It presents a pipeline capable of assessing Puppet manifests for absent or incorrect ordering constraints between puppet tasks omitted during installation, as well as potential oversights in the proper linking of software, designated as notifications in the paper. By scrutinizing the original Puppet manifest and constructing a dependency graph from the program trace, the paper identifies potential missing dependencies. This aspect remains unchanged in our pipeline. However, our approach diverges significantly with the introduction of a mutation process and state checking, features not addressed in the original paper. Additionally, we are contemplating leveraging their implementation of the trace primary building block representation (FStrace) with minor adjustments tailored to our specific requirements. We still have to come up with a way to properly obtain the trace and for that we will further investigate how it was done in this paper and the first paper. identify (1) resources that are related to each other (e.g., operate on the same file), and (2) resources that should act as notifiers so that changes are correctly propagated Puppet captures all the ordering relationships defined in a program through a directed acyclic graph and applies each resource in topological ordering

3.3 Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management

This paper presents a technique to test an IaC program end-to-end, which works on Kubernetes. The idea of this technique is to test the IaC during execution by checking intermediate states and confirming if they are the desired states. This might be very important for our approach since with this tool, we can add intermediate operations to test intermediately if these operations can lead to an undesired state or we can test the transitions during execution on different states to test for vulnerable starting states to then find vulnerabilities in the modules. Although Kubernetes is not the IaC we may be testing the approach and technique if it can be adapted for Puppet.

3.4 Automatic Reliability Testing for Cluster Management Controllers

This paper presents a tool for automatic reliability-testing on cluster-management controllers and an automatic reliability-testing technique for state-reconciliation systems. Cluster-management controllers refer to the controllers in Infrastructure as code tools, such as Kubernetes. Additionally, the paper leverages the state-centric interfaces. This means that a controller's actions are a function of its view of the current cluster state and test the controllers by introducing state perturbations. These systems differ from the IaC on which we will be testing the modules, but the state-reconciliation quality is a very important property in this paper as well. Our project also aims to test for reliability, which is why we might incorporate the testing technique into our work. Additionally, the perturbations of states method might be useful for our project.

3.5 Testing Practices for Infrastructure as Code

Continuous deployment (CD) is the process of rapidly deploying software or services automatically to end-users. This paper gives us the best practices related to testing infrastructure as code used to automate CD. It extracts the top Google searches on the topic and summarises them in the paper. It introduces the notion of open coding for the crawling of new text resources to justify their qualities, which consists of an experimented person reading each file and sorting them into categories cross-checked by two others. From this they extract 6 main practices, 5 of which

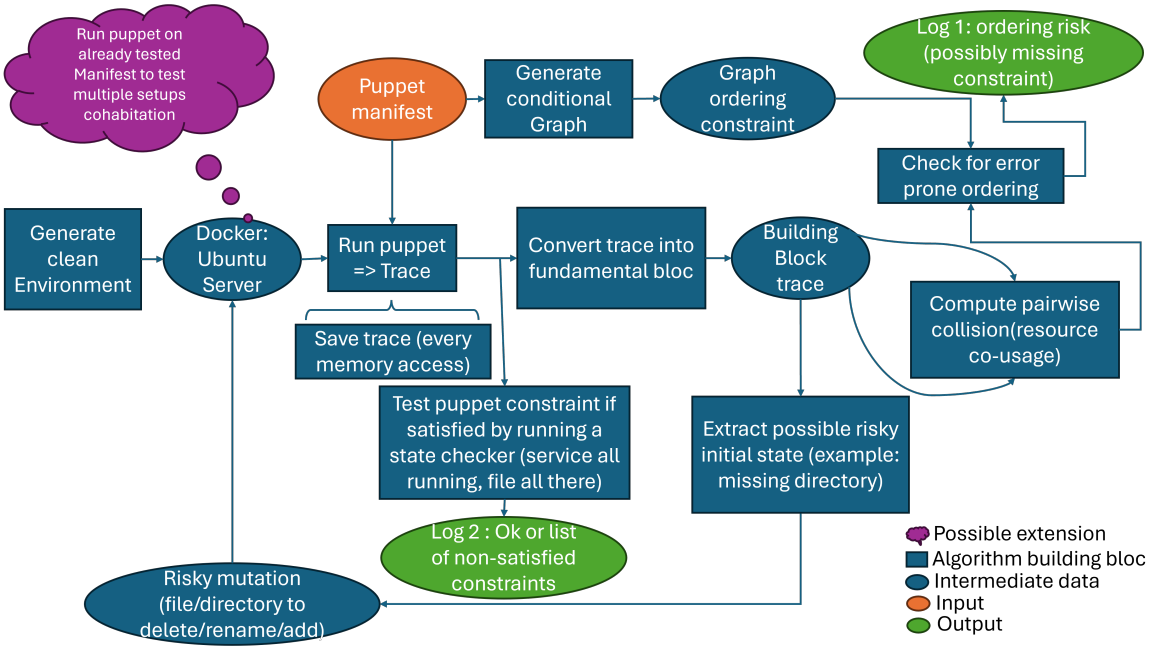

Figure 1: High-level view of the pipeline

are implemented in some way in our architecture: Use automation; Sandbox testing; Test every IaC change; Behaviour-oriented test coverage; Avoid coding anti-patterns. The remote testing practice isn't relevant to us in the context of Docker emulation.

4 Approach

[Figure 1] Our testing approach for Puppet manifests focuses on ensuring accurate execution and reliability. To begin, we initiate our process by deploying a new and unmodified instance of the Ubuntu puppetserver within a Docker environment. Ubuntu server, based on Debian Linux, is chosen due to its widespread usage and, according to the official Puppet website, high compatibility with Puppet modules, and the puppetserver is a base docker image provided by Puppet. The second building block of our architecture involves mutating the OS according to one of the risky mutations contained in the list of risky mutations. The list is obtained from a later block in the architecture (closed loop over our pipeline). If there are no risky mutations generated up until this block, like for example on the first run, the program will skip this block. The third building block of our architecture involves executing the Puppet manifest and receiving important outputs: trace, catalog and the resulting system, from which the directory representation of the state is saved for a later block. The first time this block is run, the generated catalog then gets used in the block "Generate conditional Graph" for generating a graph of the dependencies in the manifest. The "Convert trace into fundamental blocks" block is responsible for disassembling the generated trace and extracting the system resources manipulated by Puppet during its execution including details like directory links, file names being modified, or any other resources the manifest interacts with. This information is crucial for identifying potentially risky mutations, which are then added to a list for further processing. A risky mutation, for example, would be to delete a directory targeted by Puppet to create a file, and then observe if Puppet still converges. Identifying these risky operations is the task of the "extract possible risky initial state" block, which generates outputs that are passed to the beginning of the pipeline. Once the Puppet execution and the conversion to building blocks of the trace are complete, we have two blocks for processing the final output : The first block, "Check for error-prone ordering," assesses the existence of ordering constraints within the manifest. For instance, it verifies if memory reads occur after writes to the same region. These constraints are extracted from the Puppet manifest's constraint graph, contributing to one aspect of our output. The graph is constructed according to [Practical Fault Detection in Puppet Programs] using the compiled version of the manifest, the catalog, which has all the dependencies. The second block focuses on basic state reconciliation checks, which is done in the block "Test puppet constraint if satisfied by running a state checker" which is the

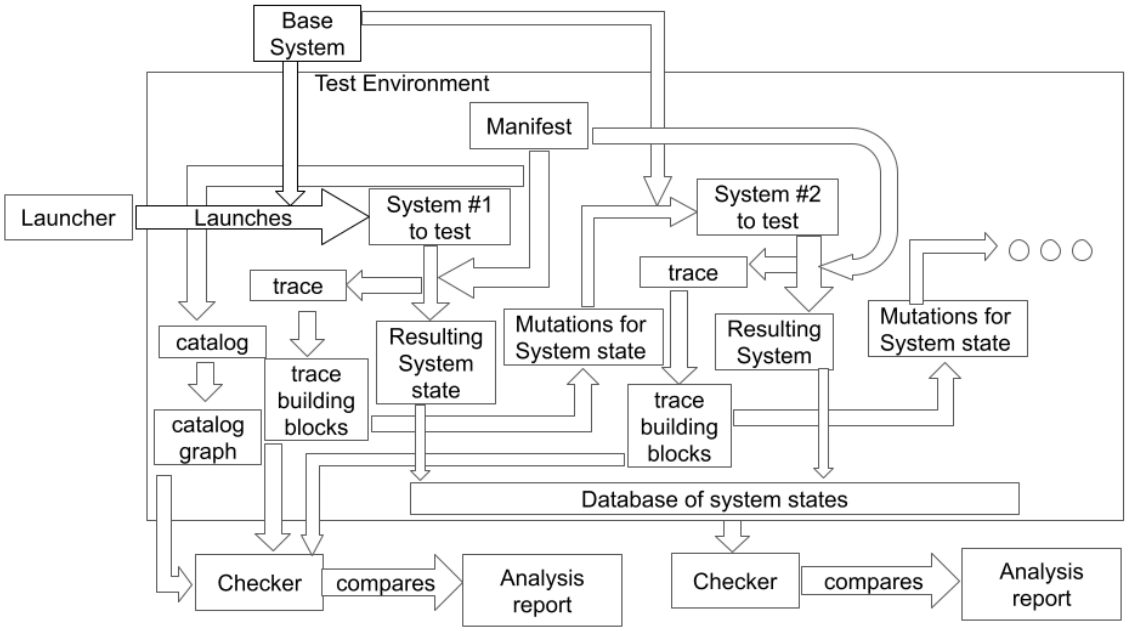

Figure 2: Architecture of the test.

state checker block. The states that are used are the ones gotten at the current run and compared to the first run that was run without mutations. By comparing the states of instances generated from different executions of the same manifest on mutated OS environments, we can pinpoint any discrepancies in the edited resources specified by the Puppet manifest. Overall, our approach enables comprehensive testing of Puppet manifests, ensuring not only correct execution but also resilience to potential system mutations and errors.

5 Implementation and experiment

5.1 Implementation

Reminder: Our code is available on GitHub at: https://github.com/theoducrey/eth_asf_proj Our pipeline implementation consists of several Python scripts ['main_one_run.py', 'manifestGraph.py', 'risky_mutation_generation.py', 'run_docker_puppet.py', 'state_checker.py', 'trace_analyzer.py', 'trace_handling.py', 'out_put.py'], two Docker files ['Dockerfile', 'docker-compose.yml'] and a JSON file containing the configuration of the manifests we want to test [available_manifest_param.json'].

Our architecture relies directly on the Puppet Forge server to pull the targeted Puppet module, making the addition of new manifests as simple as specifying the 'module_name', 'version', and 'configuration' (the content of the manifest used). We primarily relied on the demo of each module without specifying a specific configuration.

The starting point of our application is 'main_one_run.py'. This script takes two arguments: the name of the entry to read from the configuration file and the number of iterations (mutation runs) to which we want the manifest to be subjected. The execution follows these steps:

Initialization

1. **Creating Pipes and Master Lock**: The first step is to create the pipes used to model the flow between each building block of our pipeline and a master lock. This simplifies the use of our pipeline in a multi-threaded architecture. The 'main.py' file can serve as the starting point for a multi-threaded implementation.

2. **Preparing Building Blocks**: Each building block is prepared for its task by assigning its respective pipes, logger, and additional arguments, including the catalog obtained from the manifest by running it on a clean system.

3. **Manifest Graph Generation ('manifestGraph.py')** : The catalog is a JSON file, but we need a way to quickly query if a relationship exists between two resources in the manifest. For this, we convert the JSON file into a NumPy matrix of relationships. Each resource is assigned a row and a column; if a precedence constraint exists between resource 2 and resource 11, then matrix[2, 11]

= 'B'. Other relationships such as 'require' and 'notify' are similarly encoded, require 2 for 11 == matrix[2, 11] = 'B' ; notify 2 in 11 == matrix[11, 2] = 'N'; subscribe 2 to 11 == matrix[11, 2] = 'N'. The existence of a relationship can be easily checked using the getter method 'edge_res1_res2', which takes the names of the two resources as arguments.

Main Run

1. **Running Puppet and Mutations ('spawnRunPuppet.py')** : We implemented the process of running Puppet and handling mutations using a class called SpawnRunPuppet. As with each class in this project, the first action of the entry point method process_mutation_queue() is to wait for data in the form (run_id, mutations_list) from its input pipe. Upon receiving the data, the method performs several steps. First, for each mutation, it generates its Linux terminal equivalent command. Next, it starts a fresh instance of the official puppet-server Docker image. It then creates a directory for the results, which is automatically linked to the local Python project directory. Following this, the configuration file is read to override the manifest on the Docker image with the one specified in the configuration file, and the corresponding module is installed using puppet module install. The output mode of Puppet is set to debug to obtain more detailed information in the trace. The manifest is then applied using Puppet, attached with strace to capture all syscalls performed by the application on the OS image, filtered by a list of syscalls (syscall_filter). The results are extracted and saved in the result folder. Finally, the Docker image is shut down and cleaned to ensure no interference with the next run.

2. **Trace Handling ('trace_handling.py')** : The next step was to convert the strace file obtained from running the manifest into a format processable by other modules. To achieve this, we read the file line by line and separate resources using the debug prints generated by puppet apply. We implemented a multi-buffer system to handle unfinished operations in the trace that span multiple lines. The core of the implementation is a switch case that parses each syscall and finally saves the files touched by the manifest along with the impact of the corresponding syscall on them. Additionally, we implemented a basic file descriptor table to keep track of the correspondences between paths and file descriptors. To finalize the function, we use the list of renamed files to retain only one version in the final output. The rationale behind this is that our pipeline focuses on warning about potential side effects between modules and should not emphasize order which would be a side effect of taking into account different names for the same file at different times.

3. **State Reconciliation ('state_checker.py')** : This module is responsible for testing state reconciliation. The files we mutate are those directly impacted by the manifest, and the original OS is strictly equivalent between runs. Therefore, we use the output of the tree command to examine differences between each file system (resulting pair of OS at the end of two separate runs) to the first run without mutations. If the states are correctly reconciled, the directory edges to files should be equivalent. We achieve this by parsing the tree command output into a set of edges from directories to files, then we comparing the two sets and saving the differences from the first to the second, basically comparing the pair of state of the file system to each other. Puppet and ruby-specific files that change each run, due to how puppet works, were filtered out, as well as only looking at a specific subset of file types, that we can expect will be the result of the manifest.

4. **Trace Analysis ('trace_analyzer.py')** : This part of the module is strongly inspired by the paper [Practical Fault Detection in Puppet Programs]. We are looking for common files between pairs of resources in the manifest. Then, for some pairs of operations, we check if one resource is consuming a file and another is creating it. We ensure that a relationship exists in the manifest graph between these two resources. To perform this test, we simply call the getter of 'manifestGraph', which then tells us if the manifest graph has a path from the resource that is supposed to run first to the one that is supposed to run second, according to the defined dependencies. Since the graph is a tree, it means that the first resource will run first if there is a path from the first to the second resource.

5. **Risky Mutation Generation ('risky_mutation_generation.py')** : To create new mutations, we read the

Figure 3: Normal run.

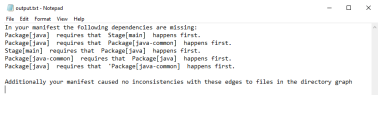

Figure 4: Added faulty dependency test.

Figure 5: Randomly added file to runs.

processed trace and retrieve the list of files from it. Depending on the action performed by the manifest on each file, we allow certain mutations to occur. For the initial version of our pipeline, each resource is considered separately. This means we are not removing, creating, or renaming files for two different resources simultaneously. Instead, we generate a set of mutations for each resource. We take a random subset of random size from all the files touched by the resource. For each element in this set, we assign weights to each possible mutation based on the type of operation performed on it by the manifest. Additionally, there is a probability that a file will not be mutated at all. We then update the weights to restrict possible mutations to only those that were not already performed on the file. According to these weights, we sample a random operation to perform on the file. Both the constraints—that mutations are only applied to one resource at a time and that the same mutation is not performed twice on the same file—are interesting aspects to consider changing in future versions of the pipeline to account for more concurrent effects.

6. **Output Parsing (‘out_put.py‘)** : Finally we parse and summarize the result of the different runs of the manifest to a more human-centered representation of possible warnings. The two outputs are for the check of the dependencies and the check for state reconciliation. If the manifest runs correctly, both of these fields should be empty.

5.2 Experiment

5.2.1 Experimental Setup

For the setup of the experiment to test a manifest, we first needed to input the content of the manifest we want to test, alongside the name of the module and version to pull from puppet forge into $availible\_manifest\_param.json$, for that the puppet page https://forge.puppet.com/ can be used. Then we are ready to launch the pipeline by executing python main_one_run.py -tm ’dictionary key in configuration’ -nr ’nbr_of_mutation_run’. For example, we could run python main_one_run.py -tm apt -nr 9. By default, it will run the Java module with 10 iterations. The run will be interrupted if there are no more mutations to do or if the maximum number of iterations is reached. We need to be aware that docker is required to run the pipeline and on something like Windows, may need to be launched in advance. We tested our pipeline on both Linux and Windows systems. The dependencies that we tested are defined in $trace\_analyzer\_5.py$ in the list $before\_after$, and can be extended.

5.2.2 Experimental Result

With the correct dependencies, that we defined, and the state comparison, to ensure all the files we are testing for are at their correct place, while the starting state was mutated. The popular manifests that we tested, such as java, stdlib, apt, and more, performed correctly and we did not find missing dependencies and found that state reconciliation was upheld. This means that at the end of the *output.txt* we got two lines separated by an empty space saying that in both checks nothing wrong was found, this was what we expected, and also what we got.[Figure 3]

Additional tests were conducted on if it could find wrong mistakes in dependencies if we for example test for an accessed-accessed dependency. When a file is accessed from two different resources, it doesn’t imply that one has to come before the other, but if our tool works, it needs to output in the first part of the output file that missing dependencies were found in the manifest. When testing this is what we got, ”missing dependencies” between resources were found.[Figure 4]

Further testing was done on the state reconciliation part to see if it can detect file-directory differences between states. By sometimes adding a file $text_maybe.txt$ at the end of a puppet run we created an inconsistent directory-file edge which should then be visible in the second part of the report. When we tested, in the output file second part this file then appeared as expected.[Figure 5]

6 Conclusion

The approach taken in this work was shown to work and is a viable tool for testing manifests, proving that the popular manifests we tested,

were indeed handling the dependencies correctly and having state reconciliation.

The approach still has a lot of improvements possible. Because there are two comparisons, but they are done separately, a viable improvement could be to connect these two checks. Another improvement would be to add more dependency testing for the catalog graph - trace comparison to test for more dependencies not accounted for yet. Additionally, increasing the amount of file types, that are checked for may increase the thoroughness of the state reconciliation test. Also, there is a possible improvement in the mutations to make them more complicated, so that they could trigger the previous improvements to dependencies testing. The manifest takes a long time to be applied to the docker image so some work could be done in saving more intelligently the state of the docker image to respawn at a target point. Also, the pipeline can be easily made multi-threading compatible to perform run of the different building blocks simultaneously, all this work was partially explored during our work, but still needs exploration and trying to function reliably.

In conclusion, we built the first python pipeline to test puppet manifest, which relies solely on the output of the strace and tree commands, and can be mostly reused to test architecture as code in general.

References

[1] Oliver Hanappi, Waldemar Hummer, Schahram Dustdar (2016) Asserting Reliable Convergence for Configuration Management Scripts, ACM DIGITAL LIBRARY. Available at: https://dl.acm.org/doi/10.1145/2983990.2984000 (Accessed: 26 March 2024).

[2] Thodoris Sotiropoulos, Dimitris Mitropoulos, Diomidis Spinellis (2020) Practical fault detection in puppet programs, ACM DIGITAL LIBRARY. Available at: https://dl.acm.org/doi/10.1145/3377811.3380384 (Accessed: 26 March 2024).

[3] Thodoris Sotiropoulos, Dimitris Mitropoulos, Diomidis Spinellis (2019) Detecting Missing Dependencies and Notifiers in Puppet Programs, ARXIV Available at: https://arxiv.org/abs/1905.11070 (Accessed: 26 March 2024).

[4] Gu, J. T., Sun, X., Zhang, W., Jiang, Y., Wang, C., Vaziri, M., ... & Xu, T. (2023, October). Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. Available at: https://dl.acm.org/doi/pdf/10.1145/3600006.3613161 In Proceedings of the 29th Symposium on Operating Systems Principles (pp. 96-112). (Accessed: 26 March 2024).

[5] Sun, X., Luo, W., Gu, J. T., Ganesan, A., Alagappan, R., Gasch, M., ... & Xu, T. (2022). Automatic reliability testing for cluster management controllers. Available at: https://tianyin.github.io/pub/sieve.pdf In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22) (pp. 143-159).

[6] Mehedi Hasan, M., Ahamed Bhuiyan, F. and Rahman, A. (2020) Testing practices for infrastructure as code, ResearchGate. Available at: https://www.researchgate.net/publication/346749959_Testing_practices_for_infrastructure_as_code (Accessed: 26 March 2024).

[7] (2024) Linux man page Available at: https://man7.org/linux/man-pages/man3/fchmodat.3p.html (Accessed: 23 May 2024).