National College of Ireland

Detecting and Mitigating AWS-Specific Code Smells in Ansible Infrastructure as Code

MSc Research Project
MSc Cloud Computing

Srikanth Konka
Student ID: x23245824

School of Computing
National College of Ireland

Supervisor: Sean Heeney

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Srikanth Konka

Student ID: x23245824

Programme: MSc Cloud Computing **Year:** 2024-25

Module: MSc Research Project

Supervisor: Sean Heeney

Submission Due Date: 24 April 2025

Project Title: Detecting and Mitigating AWS-Specific Code Smells in Ansible Infrastructure as Code

Word Count: 9453 **Page Count**: 20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: K.Srikanth

Date: 24 April 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Detecting and Mitigating AWS-Specific Code Smells in Ansible Infrastructure as Code.

Srikanth Konka
x23245824

Abstract

This research examines the vital problem of code smells in Ansible infrastructure-as-code (IaC) scripts specifically for AWS deployment scenarios. The research uses previously studied assessment methods in IaC quality assessment to create a new three-tiered framework for code smell detection and mitigation that integrates static analysis, program dependence graph (PDG) analysis and deep learning (DL) approaches. The method required developing multiple functional Ansible playbooks targeting AWS infrastructure deployment tasks followed by code smell pattern identification and the establishment of an advanced detection solution. The PDG analysis enables the framework to detect variable dependencies, and code smells and utilizes Multi-layer Perceptron (MLP) neural networks to recognize contextual code smells. The framework categorizes code smells into six primary types (UR1, UR2, UO1, UO2, HP1, HP2) plus security vulnerabilities, all of which affect AWS infrastructure code quality and security. Experimental evaluation included testing the framework using open-source Ansible repositories and creating test playbooks focused on AWS deployment with different roles. The unique patterns of smells found in AWS-specific infrastructure code extend previous findings and hard-coded credentials and improper error handling appear most frequently. The combined PDG-MLP detection produced higher accuracy at spotting code smells compared to traditional static analysis. The research presents both theoretical AWS code smell knowledge about Ansible infrastructure along with practical tools for practitioners including a detection optimizer and solution database.

Keywords — Ansible, IaC, program dependence graph (PDG), neural networks, AWS

1 Introduction

1.1 Motivation and Problem Background

Infrastructure as Code (IaC) has totally changed infrastructure management, bringing new computing infrastructure to life by allowing engineers to define this infrastructure using machine-readable definition files rather than manual procedures (Guerriero et al.; 2019). This change has brought in software engineering best practices such as version control, automated testing, and continuous integration into infrastructure management, and as a result, has made these systems more dependable and scalable (Arachchi and Perera, 2018). Thanks to its agent-less operation and YAML-based syntax, Ansible has become one of the top IaC solutions in 2023, which can be used for automating AWS cloud platform deployments (McKendrick, 2023).

As more and more businesses are using Ansible to automate their AWS infrastructure, the quality of Ansible scripts is getting more important. Ansible code of poor quality can result in inconsistent systems, security flaws, and expensive maintenance. Code smells, i.e., indicators of future problems, have received less attention in IaC languages like Ansible than their counterparts in traditional programming languages (Carreira et al., 2025). The specialized implementation patterns found in AWS-specific Ansible code also mean that different error profiles may arise in contrast to general-purpose Ansible code, making it difficult to detect code smells (Chiari et al.; 2024).

Opdebeeck et al. (2022) found six variable-related smells in Ansible roles due to lazy evaluation of expressions in templates and variable precedence rules. However, this work did not include a thorough remediation strategy nor a specific study of cloud-based deployments. As businesses are migrating to AWS cloud platforms at an increasing speed, targeted analysis of code smells in AWS-specific Ansible code and the efficient remediation methods to fix them are critically required.

1.2 Problem Statement

Fakhouri et al. (2024) reported that AWS dominates the largest market position in the cloud platform, with about 40% total market share during 2023-24 fiscal year. AWS is used by organizations to build complex infrastructure, which they automate using Ansible as the tool of choice. The AWS-specific Ansible code is a large platform that has become a serious quality problem that must be solved. There is a limited research regarding code smells within IaC and most research focus on generic patterns without providing details for qualitative aspects unique to AWS-based Ansible code. When you start to expand your cloud infrastructure, most organizations face extreme operational issues, because what you first imagine as some small quality problems, turn into major support difficulties later. While external tools analyzing Ansible code have little use, they're helpless in contextual dependencies, and can't look past simplistic code patterns. On the other hand, the implementation of deep learning (DL) approaches can potentially enhance the detection accuracy.

There is still a considerable disparity between the theoretical examinations of code smells and useful advice accessible to practitioners (Garousi and Kucuk, 2018). Therefore, Ansible developers need to work on certain ways to eliminate the creation of unpleasant code smells and achieve effective code refactoring. The data shows that there is a need for extensive research that ties the analytical knowledge around the flaws that occur in code to AWS-specific Ansible code quality assurance.

1.3 Research Question

What benefits can be gained by Ansible practitioners with the use of DL models to effectively identify and mitigate code smells in their AWS-focused infrastructure-as-code (IaC) implementations to improve maintainability and reliability?

1.4 Research Objectives

The following specific objectives are attended to by this research:

- Create and test various Ansible playbooks for AWS to find out which AWS implementation-specific code smell patterns appear in managing AWS infrastructure.
- Draw an extensive methodology to detect AWS-specific code smells in Ansible that has an integrated approach using program dependence graph (PDG) analysis, and deep learning (MLP) techniques.

- Establish the theoretical knowledge and provide practical guidelines to assist developers with prevention and correction of the identified code smells within the AWS targeted Ansible operations.
- Develop a framework that can be used to provide actionable feedback on AWS infrastructures for practitioners using Ansible and validate them via an implementation of a tool.
- Test experimentally how well the suggested detection and mitigation approaches compared to existing methods work.

1.5 Research Contributions

Specifically, this research makes the following contributions.

- A comprehensive catalogue of code smells specific to the use of AWS in Ansible IaC scripts, aside from the already documented general Ansible code smell catalogues.
- A novel three-tier research methodology is presented that makes use of static analysis, PDG analysis, and MLP based analysis for accurate code smell detection.
- Provision of detailed refactoring strategies and best practices strictly related to removing AWS-specific code smells in Ansible.
- An open-source implementation that serves to demonstrate the research and be of practical value for Ansible developers.
- The empirical validation of detection metrics using MLP and mitigation effectiveness across multiple AWS-focused Ansible deployments are also discussed.

1.6 Thesis Structure

In Chapter 1, research problem, motivation, objective, and contributions are introduced. In chapter 2, related work on code smells in IaC, Ansible specific implementations, quality assurance tools, and machine learning approaches are reviewed. The research methodology with regards to playbook development, code smell analysis, framework development, and validation methods are presented in Chapter 3. In chapter 4, the system architecture, proposed integration components, and detection mechanisms are discussed. The implementation, detection models, and AWS integration parts are discussed in Chapter 5. The performance of the framework is evaluated in Chapter 6 and its implications are discussed. The conclusions and limitations of the research are presented in Chapter 7.

2 Related Work

The present research evaluates studies regarding code smells in Infrastructure as Code (IaC) particularly focused on both Ansible and AWS-specific implementations. The review includes investigations of general IaC code smells and Ansible-specific evidence and tools that check IaC quality and ML systems for code smell identification.

2.1 Code Smells in IaC

Sharma et al. (2016) analyzed code quality in IaC architectures by identifying and analysing configuration smells of Puppet scripts. They also created a catalog of 13 implementation, 11 design configuration smells, and studied 4,621 Puppet repositories. They used tools such as Puppet-Lint and their own tools Puppeteer to detect such smells across repositories. They analyzed the results and found that design configuration smells have a 9% higher average co-occurrence among each other than implementation smells and configuration smells belonging to different design smell categories tend to co-occur when measured by volume. Due to lack of available parser libraries for Puppet, they were limited in their study by reverting to string

matching instead of parsing. They showed that configuration code was as important to treat the same was as production code.

Based on their previous work on Puppet scripts, (Rahman et al.; 2021) replicate their results on Chef and Ansible scripts using a differentiated replication study. They qualitatively analyze 1,956 IaC scripts to identify security smells and develop a static analysis tool called SLAC (Security Linter for Ansible and Chef scripts) to automatically identify these smells within 50,323 scripts from 813 open-source repositories. Six security smells are identified for Ansible and eight for Chef, two of them defining new categories of smells (missing default in case statement' and 'no integrity check') which is not observed in their previous work. They identified 46,600 such occurrences of security smells and 7,849 hard-coded passwords. The limitations of this study involve subjectivity in smell derivation and oracle dataset construction, as the study is restricted to only open-source repositories.

A large study of security smells in IaC scripts was performed by (Rahman et al.; 2019) in the context of Puppet scripts. Their research goal was to prevent the use of insecure coding practices by practitioners who develop IaC scripts with empirical analysis of these security smells. To identify the smells, the authors methodologically conducted qualitative analysis on 1,726 IaC scripts, resulting in identifying seven security smells. From the results of the study, it was possible to deduce the existence of 21,201 security smells, 1027 of which were hard-coded passwords. The researchers submitted bug reports for 1,000 randomly chosen security smell occurrences, collected from 212 responses got from development teams. The team also noted that security smells have a long lifetime, finding that hard-coded secrets exist over nearly 98 months. One possible limitation of the study is that it is only centered around on Puppet scripts, thus possibly minimizing the ability to generalize discoveries to other IaC languages, such as Ansible and Chef.

Rahman and Williams (2019) studied properties of source code that correspond to defective IaC scripts. Further, they apply qualitative analysis on 2,439 Puppet scripts by analyzing defect-related commits of 94 open-source projects, collecting 12,875 such commits. They used 89 raters with experience in software engineering to identify defect-related commits and determine which scripts were defective. They analyze 10 source code properties that correlate with defective IaC scripts and found 'lines of code' and 'hard-coded string' to be significantly correlated. A survey among practitioners validated their findings with highest agreement on the "include" property. Based on their identified properties, defect prediction models were built with precision from 0.70 to 0.78 and recall from 0.54 to 0.67. The potential subjectivity in qualitative analysis process is a limitation of the study, due to only "fair" levels of agreement between raters. The team of (Sotiropoulos et al.; 2020) developed a model for detecting faults in build specifications IaC, where the errors relate to missing dependencies and notifiers in Puppet manifests. To identify relationships between resources that should be explicitly declared in IaC scripts, they analyzed system call traces. Through a combination of static analysis of Puppet manifests and dynamic analysis of system call traces during manifest application, they were able to detect undeclared dependencies which might lead to determinism problems. The researchers present their approach as a tool and evaluate the tool on real-world Puppet modules. The method tackles its major flaw – lacking responsibility of all IaC dependencies declaration, in order to ensure idempotence and convergence. Their approach has a limitation of concentrating only on Puppet and many false positives in more complex execution environments.

2.2 Ansible-Specific Implementations

Opdebeeck et al. (2021) analyzes semantic versioning practices in Ansible Galaxy roles and studies what kind of changes trigger certain version increments. A new hierarchical structural model for Ansible roles is designed and a domain-specific structural change extraction

algorithm is implemented to analyze over 81,000 version increments of 8500+ roles. However, their quantitative analysis showed that most Ansible role developers follow the semantic versioning format, but they do not always follow the same rules of version bumps. Most of the time, the difference between a patch and minor increment was not clear. In addition, developers located in the same application were surveyed to qualitatively explore versioning practices and a classification model was built to predict appropriate version bumps from structural changes. Another critical shortcoming of this structural model is it covers only YAML files under five role directories, missing changes in other directories. The authors further provide recommendations for role clients, developers, Ansible community and researchers to practice more consistent versioning across the ecosystem.

In this work, (Opdebeeck et al.; 2022) explore variable-related code smells in Ansible infrastructure code and their detection, lifetime and prevalence. To this end, the authors introduce a catalog of six variable-related code smells specific to Ansible, and devise a novel PDG representation suitable for detecting these smells. Their approach was evaluated on a set of 21,931 Ansible Galaxy roles comprising of more than 1.5M files. According to the research, Ansible code becomes riddled with variable smells—33.8% of roles have at least one smell and it is increasing over time. The study further reveals that code smells tend to pile up in the same files and roles, and tend to linger for extended periods in the structure of a codebase after being introduced. The approach has the limitation of not being able to deal with dynamic file inclusions and some complicated variable expressions. Its implications point to the importance of better-quality assurance (QA) in the development of IaC, where such smells can cause defects that may lead to expensive infrastructure failures.

Zhang et al. (2023) looks at how Ansible code is perceived by practitioners as containing test smells. For this research, the authors performed a study of how Ansible developers recognize and react to troubling patterns in test code. The paper discusses on how to understand the mindset of practitioners of test quality in Ansible. It focuses on the definition of "smelly" test code in Ansible projects and the ways that these perceptions support existing software engineering principles. This body of work provides order channels to the increasing body of research on testing IaC to assure quality and deployability. For Puppet, (Bent et al.; 2018) presents an empirically defined and validated quality model. They developed a measurement model for the maintainability aspect of Puppet code quality through an examination of the survey results and the existing software quality models. The metrics such as file length, complexity, number of exec statements, warnings from Puppet-lint, number of parameters, module degree, duplication and volume are incorporated in the model. This model was implemented in a software analysis tool and validated by the authors through a sequence of structured interviews with Puppet experts, and by comparison of results to assess quality as judged by the experts. The measurement model and the tool were validated to produce quality judgements that are close to experts' opinions. A key limitation is that metrics like number of parameters were less important than what was initially thought.

The four main contributions of (Opdebeeck et al.; 2024) in their doctoral thesis deal with the difficulties regarding QA of Ansible IaC artifacts. It begins with an introduction of PDG for Ansible, which represents behavior of infrastructure code in graph form so as to allow static analysis without having to execute code. Secondly, PDG is used to detect the behavioral code smells of Ansible variables that can be defect indicators. The empirical studies suggest that variable-related code smells are prevalent and persistent in Ansible code, while security smells are often associated with dataflow and control-flow indirections. Ansible plugins often depend on third-party software which makes the deployment supply chains complex.

2.3 Tools and Approaches for IaC Quality Assurance

Sobhani et al. (2021) study reproducibility related to IaC issues on Ansible scripts. A catalog of five validated reproducibility smells that includes broken dependency chain, incompatible version dependency, assumptions about environment, hardware specific command, and unguarded operation are studied. A tool called Reduse was implemented to detect these smells in Ansible scripts, and an empirical study was conducted on open-source repositories to report their prevalence and relationships. They found that broken dependency chain was the most found smell (in about 71% of the tasks they analyzed). The study also linked certain reproducibility smells with each other to significant positive correlations, showing that one smell is probable if another occurs. The limitation of their approach is that some smell instances that are detected can be false positives, since it uses heuristics which requires validation in the context of the application domain.

Dai et al. (2020) SecureCode is an analysis framework proposed by the researchers which automatically extracts and composes embedded scripts from infrastructure code and then detects risky code patterns with correlated severity levels and business impacts. An empirical study was conducted on 409 rules from ShellCheck and PSScriptAnalyzer categorizing issues based on possible impact (security vulnerability, availability, performance or reliability) and assigning severity level. The evaluation results demonstrated that SecureCode was able to identify 3,419 true issues with only 116 false positives and 1,691 of those issues had high severity levels. However, their approach has a limitation of being unable to comprehend all Ansible modules and for not being able to translate arbitrary irrelevant operations in script form into script languages.

The problem of configuration bugs in large-scale data centers and cloud computing environments is a critical problem addressed by (Shambaugh et al.; 2016). The authors present Rehearsal, a Puppet verification tool, and focus on detecting non-determinism in Puppet manifests since deterministic configurations aid in consistent behavior of testing and production environments. To make determinism checking tractable, they perform three key analyses: (1) elimination of unnecessary resources; (2) removal of side effects inaccessible to the rest of the program; and (3) commutativity checking adjusted for idempotent operations. Finally, the semantics are encoded as formulas into the SMT solver by the determinacy checker. The authors show Rehearsal can find determinism bugs on a number of real-world examples. The limitation of their approach is that they don't handle Puppet's exec resource type which runs arbitrary shell commands.

GLITCH, presented by (Saavedra et al.; 2023), is a technology agnostic framework for the automated polyglot code smell detection on IaC scripts. Specifically, GLITCH transforms IaC scripts into an intermediate representation and allows different code smell detectors to be written on top of it. The studies conducted by the authors compare GLITCH with other popular tools and show that GLITCH can decrease the effort of code smell analysis for several IaC technologies, while at the same time attaining higher precision and recall. A main contribution of GLITCH lies in the consistency it provides for code smell detection across different IaC technologies, which is especially crucial for those projects that employ multiple IaC tools. A limitation with GLITCH is that the intermediate representation is a little simple, failing to represent certain nuances of different IaC technologies.

2.4 ML Approaches to Code Smell Detection

The ML approach that (Di Nucci et al.; 2018) replicates is from a previous work that reported ML models could detect code smells with over 95% accuracy. Using the original datasets, they noted that there were instances affected only by a particular type of smell that had a clear difference of metric distribution among smelly and non-smelly code. To test the generalizability of ML techniques, they further built more realistic datasets that contain

multiple types of smells, and a production like distribution of smelly and non-smelly instances. Applying the same ML algorithms to these datasets, which were now revised to the new scale of the data source, showed significantly worse results in terms of smaller F measure scores when compared to the original study. With the help of statistical analysis, they confirmed that smelly and not smelly elements have very different distributions of their metric values, with large effect sizes that allow the classification in the original dataset.

An approach for detecting code smells is proposed by (Zhang et al.; 2022) – DeleSmell, that uses deep learning in combination with latent semantic analysis. The approach is shown to cover three major limitations in the current studies: the incomplete feature extraction, absence of unified datasets, and use of data enhancement technique such as SMOTE. To this end, DeleSmell extracts structural features and semantic features with latent semantic analysis and word2vec from 24 real world projects, resulting in more than 200,000 samples in its dataset. To resolve dataset imbalance, they proposed a refactoring tool by transforming the non-smelly code into smelly code guided by the real cases. The evaluation of DeleSmell is conducted which revealed that it boosts the accuracy of code smells detection by up to 4.41% compared to existing approaches. The approach of feature extraction and the novel data enhancement technique are highly instrumental as they make the model able to detect complex code smells that have many structural branches and recursion nested to a larger depth.

To tackle the problem where available ML models can detect only a single type of smelly in code elements, (Guggulothu and Moiz, 2020) propose a multi-label classification approach for code smell detection based on code element. The authors note that, in reality, one code element has multiple design problems and existing mechanisms cannot detect code smells by taking into account the correlation among code smells. They convert previous research's two method level Code Smells datasets (Long Method and Feature Envy) into a multi-label dataset and apply three multi-label classification algorithms. The resulting analysis shows lift measure of 1.9 for the positive correlation of Long Method and Feature Envy smells. For classification purposes, the methods which incorporate correlation achieve higher performance (average 95-96% accuracy) than the method that works with each smell independently. The authors proceed to conclude that development of tools that detect multiple code smells at the same time by assuming the correlation can help in directing developers to prioritize the critical code smells for refactoring to reduce the maintenance effort.

2.5 Critical Analysis

From the literature review, some key findings come up in connection with the code smells in IaC, particularly in Ansible implementations. (Rahman et al.; 2021) have characterized security smells in Ansible scripts and (Opdebeeck et al.; 2022) have characterized variable-related code smells. There are simply many studies which have characterized IaC-specific code smells. These smells can be automatically detected through various tools such as SLAC, Reduse, and GLITCH discussed in the previous sections.

However, significant research gaps remain. All the studies concentrated on particular IaC approaches (for instance, Puppet, mainly) and restrict the generality to associate with Ansible. Secondly, recent code smell detection using ML approaches has shown promise, e.g., (Di Nucci et al.; 2018) and (Zhang et al.; 2022), but there has been little importance to apply such techniques to IaC code smells in the AWS-specific context. There are also existing approaches that have difficulties with dynamic inclusions and expressions of variables in Ansible as stated by (Opdebeeck et al.; 2022).

To address these limitations, we work towards developing a new hybrid approach that combines both PDG and ML-based model to detect code smells in AWS focused Ansible scripts. To create a more complete detection system to detect correlated smells together, our work is based upon the feature extraction of DeleSmell and multi-label classification

techniques mentioned by (Guggulothu and Moiz, 2020). Furthermore, this approach also allows semantic analysis to cope with dynamic inclusions and complex expressions in Ansible scripts.

3 Research Methodology

As shown in Fig. 1, the research methodology utilizes a systematic approach to exploring the code smells in AWS-specific Ansible infrastructure code and proposes a means to synthesize detection and mitigation strategies for the same. It has three phases in each methodology. In Phase 1, we created a solid ensemble of Ansible playbooks to stage the AWS infrastructure management in the real-world with regards to infrastructure provisioning, server configuration and network security configuration. These playbooks made use of different implementation approaches to cover as much as Ansible features and AWS conventions. These playbooks were then evaluated using a combination of PDG analysis, ML analysis, and AWS pattern detection to spot code smells. In phase 2, a multi-tier detection framework was developed from basic statistics analysis, PDG, and sophisticated ML models. By integrating these three detection methods, we were able to identify a high number of different code smell types and arriving at targeted refactoring strategies. In Phase 3, the framework was validated and evaluated using standard metrics including detection accuracy, performance, and refactoring effectiveness. This helped us arriving at our research product: an AWS specific catalogue of code smells, a tool for detection, reporting, and refactoring guidelines.

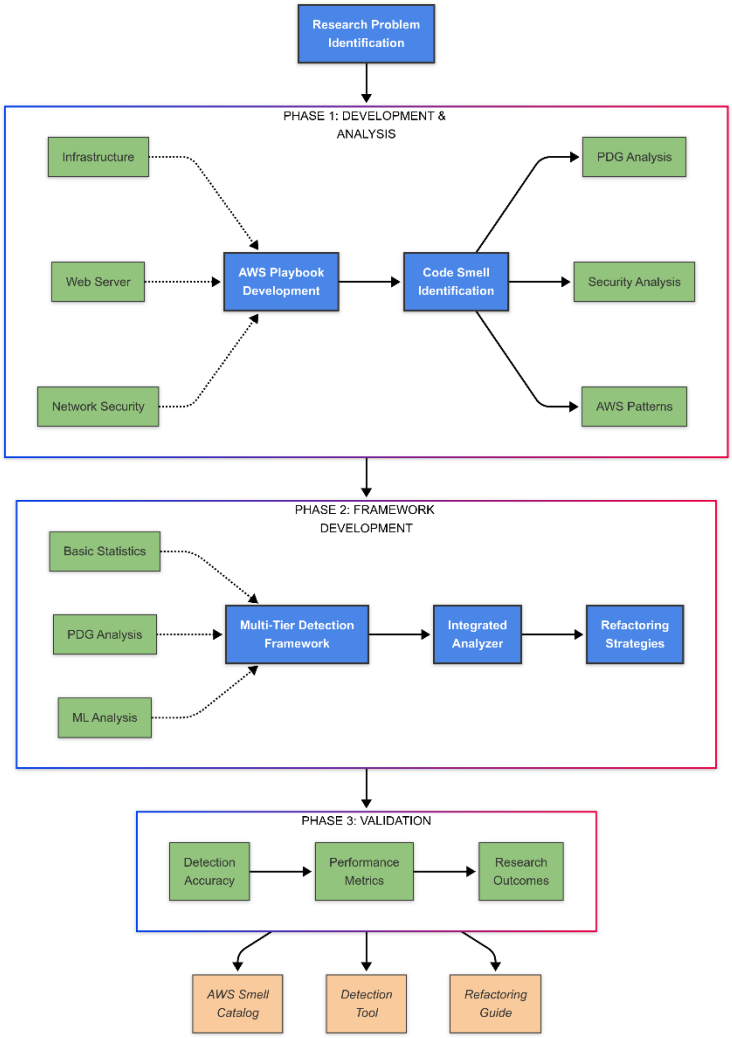

Figure 1: Ansible Code Smell Detection Methodology

3.1 Phase-1: Playbook Development

The initial phase involved developing a strong collection of Ansible playbooks that are designed to mimic typical AWS infrastructure management tasks. Three major categories of the development have been outlined – Infrastructure Playbooks, Server Configuration Playbooks and Network Configuration Playbooks. Infrastructure Playbooks contained AWS infrastructure provisioning of EC2 instances, VPCs, security groups, RDS databases, S3 buckets, and other infrastructure patterns such as auto-scaling groups and multi-region deploys. Server Configuration Playbooks dealt with the setup tasks on the AWS servers, for instance, web server setup, database server setup, and security rules for separate and clustered environment. The instructions for creating VPC, security group, NAT gateways, and VPN were all included in the Network Configuration Playbooks. For each category, we developed multiple implementation approaches covering real-world usage patterns and possible occurrence scenarios of the code smells. This made sure that the AWS conventions and the Ansible features were evaluated comprehensively.

3.2 Phase 2: Code Smell Analysis

We start off by applying (Opdebeeck et al.; 2022) PDG technique to identify the six variable-related code smells: Unsafe Reuse due to Impure Initializer (UR1), Unsafe Reuse due to Changed Data Dependence (UR2), Unconditional Override (UO1), Unused Override (UO2), Unnecessary set_fact (HP1) and Unnecessary include_vars (HP2). This set a baseline to make sure we really captured the smells identified in playbooks focused on AWS development. Furthermore, we extended our investigation to encompass security-related smells outlined by (Rahman et al.; 2021) as well as a generalized implementation of smells described by (Sharma et al.; 2016) to understand how these function in AWS context. A thorough analysis enabled us to create a complete catalogue of AWS-specific code smells going beyond similar previous research.

3.3 Framework Development

The analysis phase yielded findings that guided the development of a complete system for detecting and classifying code smells in AWS-specific Ansible code. The two main analysis approaches, PDG-based analysis and ML-based detection are combined into one framework. The PDG Component is used to detect smells related to variables, augmented to detect AWS specific variable patterns. This component analyses control and data flow relationships to detect problems with variable prevalence rules, template expression evaluation, and dependency management. The proposed implementation of the ML Component uses a Multilayer Perceptron (MLP) neural network that is trained on features extracted from both the developed playbooks as well as open-source repositories. With this approach, subtle pattern-based smells that cannot be detected by structural analysis alone can be discovered. Moreover, basic statistics gathering helps with supportive metrics like code complexity, number of tasks, variable usage patterns, and so on. These components are integrated to form a sophisticated system that is able to detect a wide variety of different types of code smells with an extensible design to add new code smell patterns.

3.4 Mitigation Strategy Development

We first examine the most prevalent AWS-specific code smells and then develop mitigation strategies for each smell type. A thorough analysis was performed of the root causes, effects, and typical appearances of the detected smells. This understanding was leveraged to formulate a set of remediation techniques focused on eliminating smells with only minimal, if any, changes to the overall functionality. For each smell type, a set of strategies to remediate the

problem were presented with before and after examples to demonstrate how the techniques could be applied. The main coverage was given to AWS-specific scripting like security group configuration, credential management, and patterns for resource provisioning. Considering the practical challenges that AWS users encounter when working with Ansible in professional environments, these mitigation strategies were implemented with practicality in place.

3.5 Validation

The last phase was testing to validate detection framework and implemented mitigation strategies. This was achieved by conducting detection analysis on the three intentionally vulnerable AWS-focused Ansible playbooks. Performance metrics are used to gauge the framework's effectiveness by measuring metrics like precision, recall, and F1-score. We presented and evaluated the refactoring of a subset of detected smell occurrences with our mitigation strategies and ensured that the functionality of resulting code remained intact. These validation tests proved to provide concrete evidence, showcasing the practical usefulness of our detection framework and mitigation strategies for Ansible users working with AWS infrastructure.

4 Design Specifications

This chapter discusses a detailed description of the proposed framework design to detect and resolve AWS specific code smells in Ansible. The system architecture with its associated framework integration support effective smell detection and remediation capabilities of the design.

4.1 System Architecture

The modular, layered design depicted in Figure 2 is designed in such a way that the system architecture is extensible and has good interoperability. It has four main layers in the architecture. A code parser is included in the data layer to translate Ansible YAML files to abstract syntax trees, a knowledge base to store code smell definitions and remediation strategies, and an AWS extension library that contains AWS context. It contains two basic components, PDG analysis component used for detecting control and data dependencies, and MLP neural network component that uses a trained model to detect patterns of code smells as part of the detection layer. The detection engine, analysis manager, and reporting module are contained in the processing layer, which manages the detection process and integrates the results. For reporting the findings, refactoring suggestions, and code visualizations, an user interface is developed. This offers a complete framework to detect and mitigate code smells in Ansible playbooks within the AWS context.

4.1.1 Data Layer

This layer consists of the code parser which converts Ansible YAML files into Abstract Syntax Trees (ASTs), which can then be analyzed further to extract information such as standalone playbooks, role structures, variable files as well as task files and templates. This component makes sure that everything that needs to be interpreted by the relevant code elements is correctly interpreted and thus available to be analyzed. The knowledge base contains a wide range of code smell definitions, detection rules, and suggested remediation practice, and it retains historical data from previous playbook runs to shorten the detection time. The AWS extension library offers AWS specific knowledge in the form of resource, implementation guidelines, and known misuse patterns to provide cloud-specific context necessary for proper smell evaluation in AWS environments.

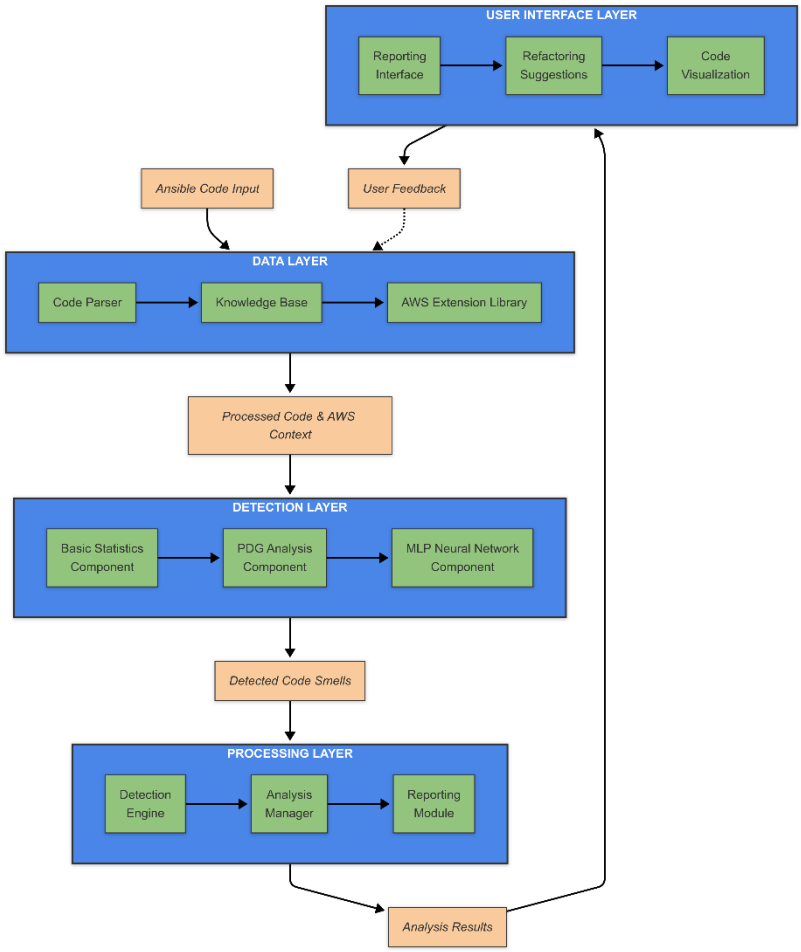

Figure 2: Proposed Ansible Code Smell Detection Architecture

4.1.2 Detection Layer

Two main components are implemented in this layer that implement the core detection functionality. The Ansible code is transformed to a graph representation by the PDG analysis component for the analysis of variable-related smells targeted on Ansible's precedence rules and template evaluation based on the dependence relations of control and data. It is especially good at finding what are usually thought of as structural and dependency issues. Further, the code elements and execution context are used as features to be input to the MLP neural network component, which uses a trained model to detect potential issues through learned patterns that may have escaped rule-based systems. Together, these components, operate both independently as well as collaboratively, to detect structural as well as pattern-based code smells.

4.1.3 Core Processing Layer

The central coordinator for the detection process is the core processing layer which oversees the execution of the entire workflow. The detection engine selects suitable analysis strategies based on code types and user preferences to orchestrate the detection process. The analysis manager merges results from detectors, resolves conflicts, removes duplicates, and processes outstanding findings according to their severity and impact. The smell discovery detail, accurate explanations, location in the code, severity, and refactoring guidelines are included in the generated reports in the reporting module.

4.2 Smell Detection Mechanism

The smell detection mechanism makes use of a two-tier approach that gradually looks through code from basic metrics to advanced pattern recognition. The first tier of this scoring system comprises basic statistics gathering, which collects such things as lines of code, how many tasks there are within the *npm* package, the number of variables, and template expressions. These metrics are analyzed using a complexity score to give context for further analysis. The second tier includes two parallel detection paths, namely, PDG analysis and ML analysis. PDG analysis produces a program dependence graph between different parts of Ansible code using data and control dependencies to detect structural code smells about usage of variable, relations between task and flow of execution. At the same time, ML analysis employs a trained neural network to perform pattern-based code smell identification using extracted features on the code. The dual approach of this framework enables both structural issues detection and detecting small patterns indicative of code smells. Finally, the results from both of these detection paths are integrated by consolidating smells, assessing severity and pairing with suitable refactoring suggestions. By taking such an approach, we make sure to detect all types of code smell in AWS specific Ansible code.

4.3 ML Model Architecture

A neural network based multi-layer perceptron (MLP) has been developed for code smell classification analysis that is made up of multiple hidden layers. The extracted features are then passed through a feed forward neural network with non-linear activation functions through its fully connected layers to help detect the possible code smells. An attention layer is used to enhance its performance through directing itself to the important code elements that make it predict better by identifying context-dependent smells. The final step is the classification layer that assigns probability to different smell categories based on patterns learnt from analyzed code segments. The analysis is used to tell apart certain/potential problems based on confidence measures.

Thus, the ML Analysis path begins with feature extraction, where the network extracts code characteristics that are relevant to smell detection and quantify them. There are structural elements, semantic patterns, and contextual information about the AWS specific code. It is implemented to get a detailed feature collection from Ansible code repositories and other synthetically generated playbooks. The code smell label identification is done through running these playbooks on PDG. These playbooks are extracted to about 28 features together with labels and this together forms a code smell detection database of about 9216 entries.

4.4 Framework Integration

The framework includes the detection paths that uses smell consolidation, severity assessment and refactoring suggestions to locate the possible code smells. It also resolves duplicate entries as well as provides cases where both approaches detect a smell. The severity assessment gauges how serious each detected smell is, with regards to security implication, maintenance burden, and operational risk. The refactoring suggestions component associates each detected smell's remediation strategies. It plugs into Ansible workflows at each stage in the development process and brings value.

During playbook development, it can be used as a real-time QA tool, where it can be included into the continuous integration pipeline as an automated quality gate. It also takes advantage of AWS services to detect AWS specific issues precisely by using resource definition and best practices to validate infrastructure configurations. The knowledge that exists in this framework with respect to AWS is an advantage that will help detect problems that standard IaC analysis tools may overlook.

5 Implementation

5.1 Framework Implementation

The implementation of the framework presented in Fig. 3 consists of several parts working together to detect and remove code smells from the Ansible infrastructure in AWS. It starts with playbook acquisition and parsing, which involves normalizing whitespaces, processing include statements, etc., and discovering the hierarchy of plays, tasks, and variables. It then extracts basic statistics about the structure of the playbook that corresponds to the lines of code, number of tasks, variable declarations and template expressions. These statistics are then used to calculate a complexity score that represents an initial review of potential quality issues.

The system represents variables, tasks, and roles themselves in the form of a PDG and dependency edges between them. The PDG analyses the variable precedence rules and template expressions to find issues of lazy evaluation. The analysis studies variable definition, modification, and usage for the purpose of identifying learning patterns such as unsafe reuse, unconditional overriding, and redundant declaration causing problematic maintenance and reliability.

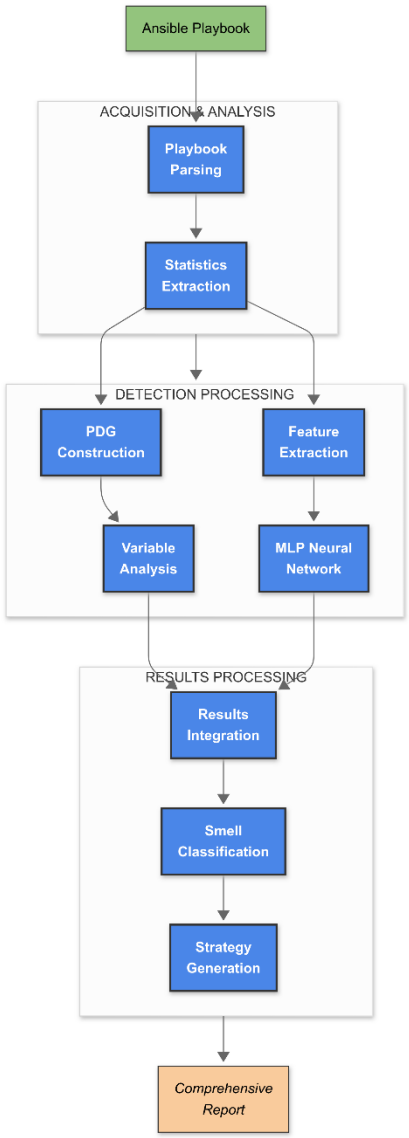

Figure 3: Ansible Code Smell Detection Framework

Special rules, and contextual analysis of the AWS resource definitions, are used to detect the AWS specific security vulnerabilities. The MLP neural network is trained over the preprocessed features, where the features are fed through multiple hidden layers with non-linear activation functions and the probabilities on each possible smell type are generated. The PDG results and ML results are then consolidated into a unified result set, and a special handling is given to the smells that the two approaches have in common. Each detected smell is subject to smell classification and severity assessment, taking into consideration the security implications, maintenance burden and operational risk in order to provide context to prioritize it.

A customized refactoring strategy is generated for each identified smell specific to AWS that are aligned to AWS best practices. These include step-by-step instructions on how to fix, before and after code samples, and explanations on how the proposed changes solve the real underlying issue. The outputs in the form of an organized report include statistical summaries, findings and actionable recommendations. The report generation provides meaning to users when it comes to understanding the overall quality profile of their infrastructure code, and what should they prioritize to improve.

5.2 Detection Models Implementation

The AWS-specific Ansible code is analyzed for code smells using the feature set generated for ML analysis. Specifically, it considers 28 characteristics of Ansible playbooks, such as the code structure, variable usage, some Ansible-specific, and AWS-specific metrics. For ML-based detection, random_usage, vars_used_multiple_times, override_count, set_fact_count, include_vars_count, aws_resource_count, security_group_count, 0.0.0.0/0_usage are some of the key features.

The multi-label classification of code smells is done using MLP-based detection component. The prepared dataset of 9,216 samples and their binary labels for the seven smell types were used to train the model. The training process used a 70:30 train-test split, following which overfitting was prevented with the use early stopping and dropout layers. Adam was chosen as the optimizer with a binary cross entropy loss function.

The PDG based detection component seeks code smells present in Ansible playbooks based on variable dependencies, control flow structures, and data flow between those structures. The *EnhancedAnsiblePDG* class creates a graph abstraction of the playbook, and through tracking how data flows through the playbook, it is able to detect variable-related smells by spotting inappropriate patterns.

This solution combines the precision of PDG analysis which is superior for issues involving structural problems such as variable precedence rules and dependencies with MLP model which excels at identifying patterns which do not have obvious structural features or relational mappings. The dual approach of this framework ensures that it can detect a large variety of code smells with high precision, which is required to account for the highly complex and diverse nature of AWS-specific Ansible code.

5.3 AWS Integration

The Ansible framework integrates with AWS infrastructure components to identify AWS patterns and vulnerabilities in the AWS infrastructure components. It ranges from EC2 Instances, Security Groups, S3 Buckets, VPC configurations, and RDS Databases. As presented in Fig. 4, there are three key components in the Ansible Environment, namely, AWS Playbooks, Detection Framework, and Refactoring Strategies. In addition, the framework determines a set of AWS resource patterns that are probably influenced by a given code smell. The handful of violations that can exist are hardcoded credentials, insecure configuration, overly permissive security groups, and improper encryption settings. The AWS extension

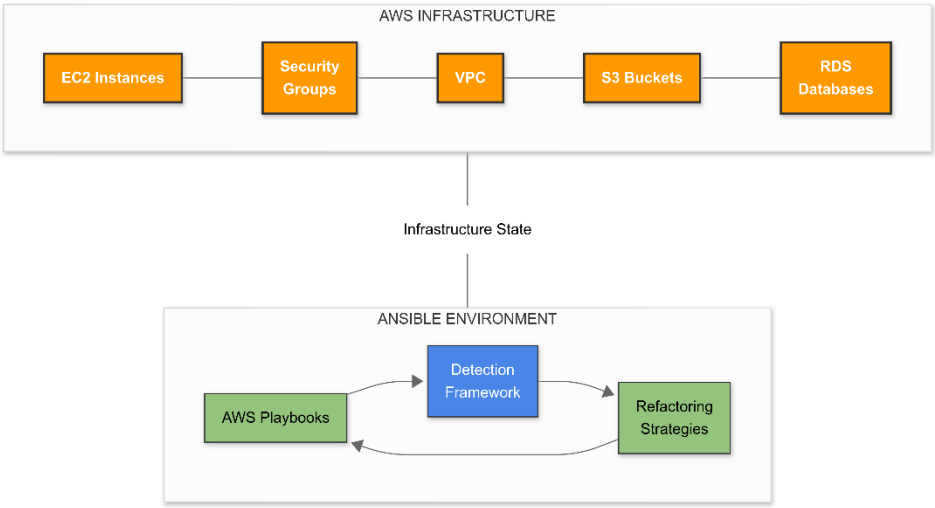

Figure 4: AWS Infrastructure for Smell Detection

library is composed of specialized configuration of AWS resource and best practice knowledge to augment detections of cloud specific issues.

This also investigates security issues related to AWS, e.g., hardcoded AWS credentials, that can be assessed with high-level of severity if it could lead to unauthorized access of the AWS account. It finds open security groups by searching for 0.0.0.0/0 in the security group rules that will create an open network for services like SSH, MySQL, PostgreSQL, and others. It also checks for missing encryption, detecting that there are no encryption parameters specified in S3 buckets and RDS instances. This approach integrates AWS-specific knowledge as dedicated features for AWS resources, custom detection rules for detection of AWS resource configurations, AWS-specific best practices in remediation suggestions, and custom checks for AWS security configuration. This helps to detect the code smells in AWS infrastructure related deployments managed by Ansible and to be able to remediate such issues.

6 Results Evaluation

6.1 Evaluation Methodology

To validate the effectiveness of the approach, this framework was evaluated by means of three vulnerable AWS deployment YAML playbooks that covers the infrastructure, security and network-related services in AWS. Such playbooks were customized, geared to host variable-related issues, security vulnerabilities and AWS-specific misconfigurations in AWS infrastructure code. As for the metrics are concerned, detection accuracy, detection coverage, source contribution, and refactoring effectiveness were evaluated for assessing the performance of framework. The technical performance of the detection algorithms along with the practical utility of the refactoring suggestions was also studied. The correlation between code complexity metrics and code smell occurrence was also analyzed to expose the corresponding behavior patterns for code development practices.

6.2 Detection Accuracy Analysis

From the Ansible playbooks analysis, it was found that a total of 14 code smells with an average of 4.67 smells per playbook were identified. Figure 5 illustrates that security smells are the most problematic (35.7%). There was also a high number of HP1 - Unnecessary set_fact (21.4%) and UO1 - Unconditional Override (14.3%). The remaining smell types, namely UR1, UR2, UO2, and HP2 formed 7.1% of the smells. Thus, this distribution underlines the significance of security-relative code in the infrastructure as it creates vulnerability on the

operational and compliance level. When examining the detection method distribution, we found a balanced contribution from both PDG and ML approaches, with PDG-only detection accounting for 35.7% of identified smells, ML-only detection also at 35.7%, and both methods concurrently identifying 28.6% of the issues. The balanced distribution lends credibility to our integrated approach and proves the supportiveness of combining structural analysis and pattern recognition in code smell detection.

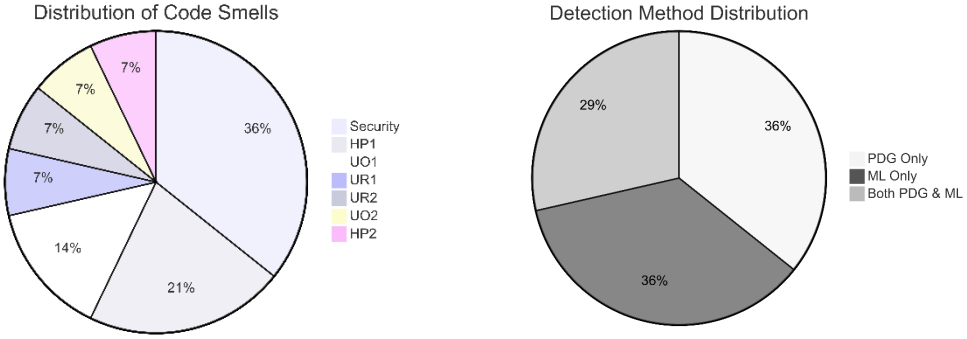

Figure 5: Distribution of Code Smells and Detection Methods

Figure 6 (a) shows the performance of the ML detection component against various smell types in the form of ROC curves. The curves for UR1, UR2, UO2 and HP2 show near perfect performance, with ROC line hugging the top left corner, implying almost perfect true positive rates along with very few false positives. The complexity is seen in detecting the UO1, HP1, and security smell type patterns as it has good but slightly lower performance. As shown in Figure 6 (b), which contains PR (Precision-Recall) curves, the model exhibits high precision for nearly all types of smells at most of the recall levels, although there is some degradation of the latter when it comes to UO1, HP1 and security smells at higher recall thresholds.

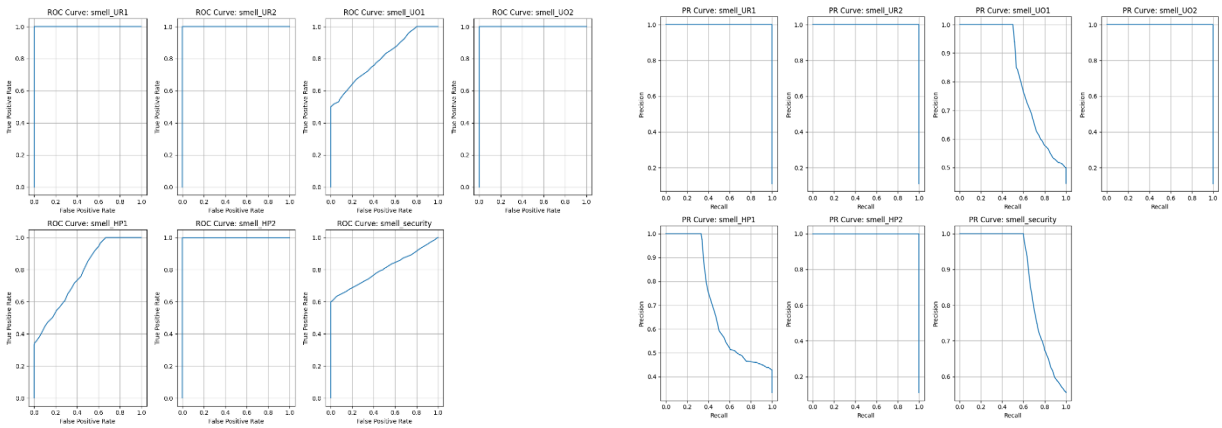

Figure 6: ML Performance for various code smell types – Left 6(a) – ROC curves – Right 6(b) – PR Curves

The loss and accuracy progress over 50 training epochs of the model training process is shown in Figure 7. The training and validation loss curve show that over several epochs the training loss decreases slowly and has plateaued at around the epoch 20, while the validation loss converges to 0.5, indicating that the model is trained without much overfitting on the training data. The train accuracy was approximately at 40% while the validation accuracy oscillates between 33% and 45%. This is reasonable for a multi label classification problem where there are seven distinctive smell types to classify. The validation accuracy curves have more variability compared to training accuracy, as is expected due to the smaller validation dataset.

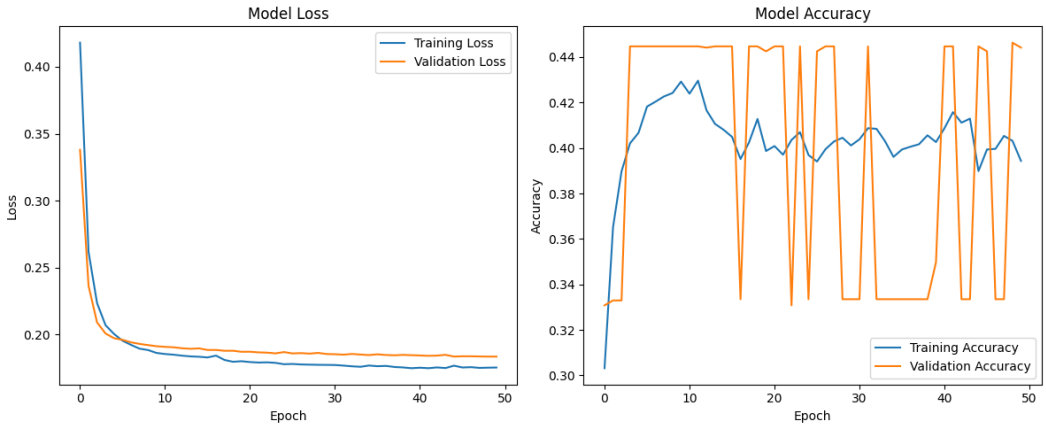

Figure 7: MLP Model Training and Validation Performance in terms of Model Loss and Accuracy

6.3 Correlation Analysis

It was found that there exists a strong correlation between code complexity and existence of code smells in AWS infrastructure playbooks as shown in Figure 8. Of the three playbooks, the most complex playbook had 9 code smells on it, followed by the medium one with 3, and finally the least complex had only 2. It indicates that with the increasing complexity of AWS infrastructure code, it is likely to have code smells and other issues. The smell count's relationship with the complexity level is non-linear and the smell count increases disproportionately at higher complexity levels. Thus, there are implications for how one should develop infrastructure on AWS, and in the case of Ansible, how one should write Ansible playbooks, suggesting that keeping simplicity and modularity would effectively minimize the presence of code smells. The complexity calculation assists in identification of playbooks that would potentially need refactoring to avert the development of code smells. Those complex playbooks should be considered to be decomposed into smaller pieces and to have continuous code reviews for high complexity areas.

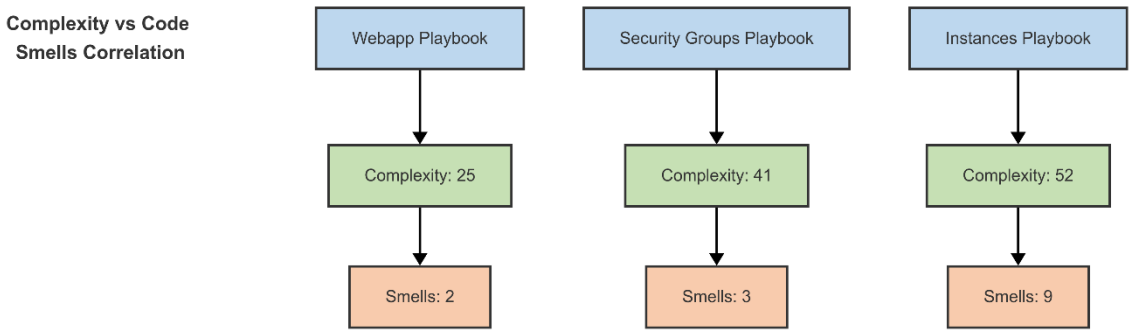

Figure 8: Code Complexity vs Smells Correlation

6.4 Findings from PDG-MLP Combined Analysis

The combined PDG-MLP analysis for each playbook reveal code smells and their specific manifestations. The *deploy_vulnerable_instances.yml* playbook has 9 code smells of which most critical comprised of security-related issues like hardcoded AWS credentials and passwords in user data scripts. The complexity scores of three such unnecessary set_fact tasks were UR1, UR2, and UO1 were found to be high.

While executing the *deploy_vulnerable_security_groups.yml* playbook, there were three code smells with a slightly higher severity while the most prominent issue is a security vulnerability pertaining to highly permissive security group rules. It was also found that there

was one UO1 and UO2 smell for unconditionally setting vpc_id and aws_region in task vars. The *deploy_vulnerable_webapp.yml* playbook received only two code smells and the complexity rank of each was 25.

Based on the analyzed code smells and as a way to initiate the mitigation of those, the exhibited code examples were presented with refactoring suggestions for each of the smells with something like moving include_vars contents to vars files and securing credentials using Ansible Vault in the case of security smells. Finally, the playbooks, the lines of code, the amount of tasks, and the usage of the variables were correlated with scores and smell counts based on their complexity.

6.5 Discussion

The evaluation of the use of code smell detection for Ansible playbooks reveals the following. As per the results, the hybrid approach PDG-MLP performed better than the stand–alone PDG analysis or stand–alone ML detection. For variable-based smells, PDG analysis proved to be better, while ML detection was better in finding pattern-based issues and security concerns. The security smells (35.7% of the total issues) indicate that security considerations are crucial while developing the AWS infrastructure code. Based on the correlation between code complexity, and the prevalence of smells, complexity management is significant to sustain high quality infrastructure code. Playbooks can be simplified and follow modular design principles in order to reduce some code smells.

During the analysis of AWS infrastructure smell detection framework, we find several patterns of code smells such as security group vulnerabilities, credential management issues, and problems in resource configuration that are AWS-related. The attack surface of AWS resources is then expanded due to security group vulnerabilities, including the use of overly permissive rules for ports like SSH. Hardcoded AWS access key and secret key credentials management are typical security issues. Insecure deployments and data breaches could result from EC2 instances which haven't even had security groups applied or sensitive data that haven't had encryption applied contributing to poor resource configuration. We also noted that carelessly dealing with AWS-specific variables for AWS resource name and identifier lead to a potential discrepancy when naming these resources. We observed that complex resource types like ec2_instance and ec2_security_group more often have code smells.

The code smell types and patterns that were detected were heavily influenced by the AWS specific context requiring specialized analysis techniques. It is shown that the framework is useful in practice and provides meaningful advice to Ansible practitioners in terms of how they can leverage Ansible to tackle specific management tasks, simply by invoking its refactoring suggestions.

7 Conclusion and Future Work

In this research, a novel framework for detecting and rectifying code smells in AWS-focused Ansible IaC is proposed and implemented. The framework achieves comprehensive detection of various code smell types by combining the PDG and MLP techniques. The research makes several important contributions to the field. Beginning with general Ansible smells, it first creates a taxonomy of AWS–specific code smells in Ansible. A two-tier framework is developed that employs PDG analysis and deep learning to detect code smells with high accuracy. Uniquely, not found in most of similar research works, this presents detailed refactoring strategies for each smell type to be used in practice by Ansible practitioners with AWS infrastructure. Additionally, an approach for measuring code complexity and a theoretical basis for how code complexity affects smell prevalence is also presented. On a broader scale, the results show great detection accuracy across various AWS deployment

playbooks; security smells are found to be pervasive (35.7% of all smells), combined PDG and ML achieve the best results (each contributes to the detection), and increased code complexity is positively correlated with higher smell prevalence.

This research framework can then be expanded as a part of future commitments to provide additional resource coverage for other services like AWS Lambda functions, ECS containers and CloudFormation to allow for greater utility beyond smaller AWS deployments. This will reduce the effort that is required for addressing detected issues manually and developing capabilities for automated application of refactoring suggestions. This would also allow creating of plugins for popular CI/CD platforms in order to integrate smell detection into the development workflow, allowing early detection and remediation.

This analysis may also extend to cover other cloud platforms like GCP, Azure etc., to determine platform-specific smell patterns and grow cross-cloud best practices. Understanding how the code smells evolve in the reality of the AWS infrastructure code shows the real-world patterns of introducing, retaining and fixing them in practice. Finally, evaluating the operational impact of various types of code smells on AWS infrastructure performance, reliability, and security will aid in prioritization of code smell remediation effort in accordance with business impacts. These extensions to future directions would increase our understanding of code quality in IaC contexts for better cloud deployments.

References

Arachchi, S.A.I.B.S. and Perera, I., 2018, May. Continuous integration and continuous delivery pipeline automation for agile software project management. In *2018 Moratuwa Engineering Research Conference (MERCon)* (pp. 156-161). IEEE.

Bent, E., Hage, J., Visser, J., & Gousios, G., 2018. How good is your puppet? An empirically defined and validated quality model for puppet. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp.164-174). https://doi.org/10.1109/SANER.2018.8330206

Carreira, C., Saavedra, N., Mendes, A. and Ferreira, J.F., 2025. From" Worse is Better" to Better: Lessons from a Mixed Methods Study of Ansible's Challenges. *arXiv preprint arXiv:2504.08678*.

Chiari, M., Xiang, B., Canzoneri, S., Nedeltcheva, G.N., Di Nitto, E., Blasi, L., Benedetto, D., Niculut, L. and Škof, I., 2024. DOML: A new modeling approach to Infrastructure-as-Code. *Information Systems*, *125*, p.102422.

Dai, T., Karve, A., Koper, G. and Zeng, S., 2020, October. Automatically detecting risky scripts in infrastructure code. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (pp. 358-371).

Di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A. and De Lucia, A., 2018, March. Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (saner)* (pp. 612-621). IEEE.

Fakhouri, H.N., Alhadidi, B., AlSharaiah, M.A., Al Naddaf, H. and Data, A.S.A., 2024, February. Critical Evaluation of the Role of Cloud Systems and Networking in the Security and Growth of the Business Market. In *2024 2nd International Conference on Cyber Resilience (ICCR)* (pp. 1-7). IEEE.

Garousi, V. and Küçük, B., 2018. Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software*, *138*, pp.52-81.

Guerriero, M., Garriga, M., Tamburri, D.A. and Palomba, F., 2019, September. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In *2019 IEEE International conference on software maintenance and evolution (ICSME)* (pp. 580-589). IEEE.

Guggulothu, T. and Moiz, S.A., 2020. Code smell detection using multi-label classification approach. *Software Quality Journal*, *28*(3), pp.1063-1086.

McKendrick, R., 2023. *Infrastructure as Code for Beginners: Deploy and manage your cloud-based services with Terraform and Ansible*. Packt Publishing Ltd.

Opdebeeck, R., 2024. *Static Analysis for Quality Assurance of Ansible Infrastructure-as-Code Artefacts* (Doctoral dissertation, National and Kapodistrian University of Athens, Greece).

Opdebeeck, R., Zerouali, A. and De Roover, C., 2022, May. Smelly variables in ansible infrastructure code: Detection, prevalence, and lifetime. In *Proceedings of the 19th international conference on mining software repositories* (pp. 61-72).

Opdebeeck, R., Zerouali, A., Velázquez-Rodríguez, C. and De Roover, C., 2021. On the practice of semantic versioning for Ansible Galaxy roles: An empirical study and a change classification model. *Journal of Systems and Software*, *182*, p.111059.

Rahman, A. and Williams, L., 2019. Source code properties of defective infrastructure as code scripts. *Information and Software Technology*, *112*, pp.148-163.

Rahman, A., Parnin, C. and Williams, L., 2019, May. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (pp. 164-175). IEEE.

Rahman, A., Rahman, M.R., Parnin, C. and Williams, L., 2021. Security smells in ansible and chef scripts: A replication study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *30*(1), pp.1-31.

Saavedra, N., Gonçalves, J., Henriques, M., Ferreira, J.F. and Mendes, A., 2023, September. Polyglot code smell detection for infrastructure as code with GLITCH. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 2042-2045). IEEE.

Shambaugh, R., Weiss, A. and Guha, A., 2016, June. Rehearsal: A configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 416-430).

Sharma, T., Fragkoulis, M. and Spinellis, D., 2016, May. Does your configuration code smell? In *Proceedings of the 13th international conference on mining software repositories* (pp. 189-200).

Sobhani, G., Haque, I. and Sharma, T., It Works (only) on My Machine: A Study on Reproducibility Smells in Ansible Scripts.

Sotiropoulos, T., Mitropoulos, D. and Spinellis, D., 2020, June. Practical fault detection in puppet programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (pp. 26-37).

Zhang, Y., Ge, C., Hong, S., Tian, R., Dong, C. and Liu, J., 2022. DeleSmell: Code smell detection based on deep learning and latent semantic analysis. *Knowledge-Based Systems*, *255*, p.109737.

Zhang, Y., Wu, F. and Rahman, A., 2023, March. Practitioner Perceptions of Ansible Test Smells. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)* (pp. 1-3). IEEE.