ADDRESSING REPRODUCIBILITY AND ENERGY-EFFICIENCY IN AI DEPLOYMENTS

by

Ghazal Sobhani

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2024

© Copyright by Ghazal Sobhani, 2024

Table of Contents

List of Tables

List of Figures

Abstract

Deploying AI models on edge devices presents challenges in ensuring reliable and energy-eficient operations. Edge AI processes data directly on devices like IoT sensors and industrial machinery, enabling real-time decision-making and reducing latency, which is crucial for applications such as autonomous driving and robotics. However, deploying these models often involves custom Infrastructure as Code (IaC) scripts, and a lack of reproducibility in these scripts can cause inconsistencies and affect system reliability. Additionally, while advancements in hardware like SoCs, FPGAs, and AI accelerators have improved Edge AI capabilities, these deployments can lead to high energy consumption.

Our research addresses these challenges through two main contributions. First, we identify and categorize reproducibility smells in IaC scripts, particularly focusing on an automation platform, Ansible, that allows imperative infrastructure configuration. We developed a tool, Reduse, to detect these reproducibility smells, in the pursuit to ensure that IaC scripts are reliable and consistent. Our empirical study reveals the occurrence of these smells in open-source projects, with significant correlations and co-occurrence patterns among them. For instance, the broken dependency chain smell was found in approximately 71% of Ansible tasks analyzed, highlighting common reproducibility issues.

Second, we comprehensively evaluate the selection of AI models on edge devices, including the Raspberry Pi, NVIDIA Jetson Nano, and Intel Neural Compute Stick. By measuring inference power consumption, accuracy, inference time, and memory utilization, we offer insights into the performance and energy eficiency trade-offs of these models. For instance, Jetson Nano provides the best accuracy at the cost of a high energy budget. Thus, our work advances the field of edge AI with the best practices in IaC, contributing to more reliable and effective AI deployments in real-world scenarios.

List of Abbreviations Used

IaC	Infrastructure as Code
SoCs	System on Chips
FPGAs	Field Programmable Gate Arrays
CNN	Convolutional Neural Network
ML	Machine Learning
DL	Deep Learning
AI	Artificial Intelligence
CI/CD	Continuous Integration and Continuous Deployment
NLP	Natural Language Processing
LLM	Large Language Models
SVM	Support Vector Machine
KNN	K-Nearest Neighbors
ANN	Artificial Neural Network
R-CNN	Region-based Convolutional Neural Network

Acknowledgements

I would like to express my deepest gratitude to Dr. Israat Haque and Dr. Tushar Sharma for their invaluable support throughout my academic journey. Their understanding and encouragement have been a tremendous help, not only with academic matters but also in navigating the various challenges faced by a new international student. Their support has been a source of great strength and inspiration.

I am also profoundly thankful to my parents, who, despite being far away, have given me the strength to persevere during dificult times. Their unwavering support and belief in me have been crucial in helping me through moments when everything seemed to be falling apart.

Finally, I wish to extend my heartfelt thanks to all the new friends I have made here. You are the warriors who have shared this battle with me, working towards creating a better life. Your camaraderie and support have made this journey a memorable and enriching experience.

Thank you all for being part of this significant chapter in my life.

Chapter 1

Introduction

The deployment and management of AI models on edge devices is becoming increasingly common as the demand for real-time decision-making and inference grows. Edge AI represents a significant shift from the traditional cloud-based AI models, which rely on centralized data centers for computation through constant data transfers between the device and the cloud station. Instead, Edge AI processes data on devices such as Internet of Things (IoT) sensors, edge servers, and industrial machinery, enabling immediate and localized decision-making [103,107]. These changes offer several key advantages, including reduced latency, enhanced privacy and security, improved bandwidth eficiency, and the decentralization of computational tasks. These benefits are essential for applications in areas such as autonomous driving, robotics, and industrial automation, where timely decision-making is needed. Recent advancements in hardware, including System on Chips (socs), Field Programmable Gate Arrays (fpgas), and AI accelerators, have dramatically improved the eficiency of Edge AI. These innovations allow sophisticated machine learning and deep learning models to operate with high speed on edge devices [23, 84].

The deployment of AI models on edge devices often involves the use of custom Infrastructure as Code (iac) scripts for provisioning and configuration. IaC automates the creation, configuration, management, and monitoring of computing infrastructure through code, typically in the form of declarative configuration specification [68, 89]. IaC offers many advantages over manual deployments, including faster, repeatable, and consistent deployment, improved scalability, enhanced reliability, and reduced operational costs [68].

1.1 Motivation

Deploying and managing AI models on edge devices presents unique challenges, often involving custom IaC scripts for provisioning and configuration. If these scripts are not reproducible, inconsistencies and errors during deployment can arise, potentially impacting the reliability of the AI models. Ensuring reproducibility in IaC scripts is critical for maintaining consistent and reliable deployments across various environments, such as development, testing, and production.

There has been some attempts to ensure the overall quality of IaC scripts. Gonzalo et al. [85] provide recommendations for effective coding practices, including maintaining a consistent style, avoiding assumptions, organizing code predictably, documenting comprehensively, and implementing error defense measures. Similarly, Kumara et al. [60, 61] emphasize the significance of reproducibility in IaC, particularly in maintaining consistent and easily reproducible environments. Despite these discussions, specific guidelines for addressing reproducibility issues in IaC scripts are still lacking. While research has explored practices impacting quality attributes such as security and maintainability [30, 61, 80, 87], reproducibility within IaC scripts remains as an important research gap.

Sustainability, i.e., optimizing the energy consumption by the deployed (ml) models, is another challenge in Edge AI deployments. Performing inference of (ml) models directly on edge devices offers numerous advantages, including minimized latency and reduced dependence on centralized cloud servers, which are essential for real-time decision-making and analysis in applications requiring immediate response. Understanding the energy consumption of ml models is essential for optimizing their deployment in edge computing environments. Knowledge of energy requirements helps developers make informed decisions about model selection, optimization techniques, and hardware configurations, paving the way for designing sustainable edge computing systems. Prioritizing energy eficiency throughout the development and deployment process enables organizations to contribute to sustainable AI solutions that align with their sustainability goals and regulatory requirements.

1.2 Research Objective

In this thesis, we address the research gap by first exploring existing knowledge about practices that affect reproducibility in IaC scripts in the existing literature through a multi-vocal literature review. We focus on Ansible scripts (also known as playbooks) because Ansible is one of the most commonly used IaC frameworks for resource orchestration and configuration [61]. The imperative nature of Ansible code offers flexibility by allowing users to specify tasks using known operating system commands but also poses a risk of creating dificult-to-reproduce scripts due to potential violations of the idempotency principle. Therefore, the first objective of the thesis is to identify practices hindering the reproducibility of Ansible scripts from both academic and gray literature sources. Additionally, we explored whether these practices are present in real-world, open-source projects and if they contribute to reproducibility issues.

Building on this foundation, the second objective of our research is to explore the feasibility of executing AI models on edge devices by identifying and addressing the challenges associated with such deployments and determining the applications that can most benefit from these models. Specifically, the study aims to compare various lightweight machine learning frameworks for deploying models on edge devices, evaluating their performance and resource utilization to find the most eficient options. Furthermore, it investigates the practical benefits and applications of AI model deployment on different edge devices, such as the Raspberry Pi, Intel Neural Stick, and Nvidia Jetson Nano, by examining real-world use cases for each of them. Additionally, the research seeks to understand how the choice of machine learning framework, model architecture, hardware platform, and optimization techniques can impact inference accuracy, computational eficiency, and the trade-offs between model accuracy and energy consumption.

By applying the insights from our IaC reproducibility research, including the development of a tool for detecting reproducibility smells, we ensure that the IaC scripts used to deploy AI models at the edge are reliable and consistent. This integration of IaC best practices with Edge AI deployment will lead to more reliable and energy-eficient deployments of AI models on edge devices.

1.3 Contribution

This thesis makes the following contributions to the field of reliable and energy-eficient AI model deployments:

- Reproducibility Smell Catalog: We introduce a comprehensive catalog of reproducibility smells, aggregating knowledge from both academic and gray literature for Ansible scripts. This catalog serves as a valuable resource for practitioners and researchers seeking to identify and mitigate reproducibility issues in IaC scripts. By ensuring reproducibility in IaC scripts, we can guarantee consistent and reliable provisioning and configuration of edge devices, thus supporting stable and dependable AI model deployments [60, 61, 85].

- Reproducibility Smell Detection Tool—Reduse: We developed <u>RE</u>pro-<u>DU</u>cibility <u>SmE</u>ll (Reduse) detector to detect reproducibility smells in Ansible scripts. Practitioners can use this tool to improve the reproducibility aspects of their IaC scripts, thereby ensuring consistent deployment environments. Researchers can utilize Reduse to explore further ways to ensure reproducibility and investigate the causes and effects of reproducibility issues. Furthermore, this tool could help maintaining the reliability of AI models deployed on edge devices by preventing deployment inconsistencies caused by non-reproducible IaC scripts.

- Empirical Study on Reproducibility Smells: Our empirical study explores the properties of reproducibility smells and their relationships with each other. Observations derived from this study enhance our understanding of Ansible scripts and reproducibility smells, providing insights into how to improve IaC practices. Understanding these relationships helps in addressing reproducibility issues proactively, ensuring reliable and consistent AI model deployments on edge devices.

- Evaluation of AI Models on Edge Devices: We assemble an extensive list of AI models and their applications on edge devices, sourced from vari-ous references. These models are classified into traditional machine learning, neural network models, deep learning, and large language models. We eval-uated these models on Raspberry Pi, Nvidia Jetson Nano, and Intel Neural Stick. By assessing inference power consumption, accuracy, processing time, and memory usage, we provide a comprehensive analysis of model performance in edge computing environments. This evaluation helps practitioners choose the most appropriate models and optimization techniques for their specific needs, balancing performance and energy eficiency.

- Edge AI Design and Deployment Guidelines: Our findings highlight best practices and potential pitfalls in deploying AI models on edge devices. By understanding the trade-offs between performance metrics under various constraints typical of edge devices, we offer detailed guidance for practitioners. This holistic approach ensures eficient and effective edge AI solutions, contributing to sustainable and reliable AI deployments.

- Replication Packages: We make our replication package, including the source code of the developed tool Reduse, scripts used to generate and analyze data, and results obtained from analyzing open-source repositories available online [9]. Similarly, for the sustainable edge AI study, we make our code available for all the model implementations and for the different models and different platforms [33]. It also includes a detailed process on how to setup and execute models on the devices.

1.4 Thesis Outline

The thesis is structured as follows. Chapter 2 and Chapter 3 covers background concepts and related works, providing an overview of existing literature and theories. Chapter 4 presents the details the research design, methodology, challenges, research questions, and evaluation of the developed tools and experiment results. Chapter 5 presents the Energy consumption measurement for inference of AI models. Detailing the research design, methodology, challenges, research questions, and evaluation of the developed tools and experiment results. Chapter 6 presents future extensions and conclusions, summarizing key findings and proposing future research directions. This concise structure ensures a clear progression through the research topic.

Chapter 2

Background

This section provides the necessary context to understand the content presented in this thesis.

2.1 IaC and reproducibility concepts

We delve into the principles of IaC and the syntax paradigms used in IaC specifications. Additionally, we discuss relevant literature that underpins this study.

2.1.1 IaC Principles

Infrastructure as Code (IaC) is a transformed paradigm in modern Information Technology (IT) infrastructure management that enables automated and scalable deployment of computing resources by treating infrastructure configurations as code [68, 81]. This approach significantly enhances consistency, scalability, and deployment speed [68, 81, 82]. The effectiveness of IaC is grounded in several fundamental principles, which we outline below:

- Idempotency: This principle ensures that applying the same configuration multiple times results in the same outcome. Incorporating idempotent properties into infrastructure code helps organizations avoid inconsistencies and achieve a predictable and reliable state [49, 60]. It is crucial for maintaining stable environments and reducing configuration drift.

- Specification is Code: This principle emphasizes that infrastructure code should be treated with the same manners as production code. Applying software engineering discipline and quality practices to infrastructure code minimizes technical debt and enhances manageability and maintainability [69]. This includes practices such as code reviews, automated testing, and continuous integration.

- Version Control: Version control is essential in IaC for tracking changes, facilitating collaboration, and allowing rollback to previous versions if necessary. Using version control systems (VCS) ensures accountability, traceability, and supports effective teamwork [28, 61, 85]. It enables teams to manage changes systematically and maintain a history of modifications.

- Reproducibility: Reproducibility refers to the ability to recreate a given environment consistently and rapidly. It is a cornerstone of IaC, motivating organizations to adopt IaC frameworks to ensure environments can be duplicated precisely as needed [24,49,61]. This is critical for disaster recovery, scaling, and testing.

- Repeatability: Repeatability in IaC advocates for automating tasks using scripts and tools, thereby avoiding manual actions. This principle enhances the reliability, speed, and consistency of infrastructure management by reducing the risk of human error [29,60,61,69]. Automated processes ensure that infrastructure configurations are applied uniformly across different environments.

2.1.2 Reproducibility Principles

Having established the core principles of Infrastructure as Code (IaC), including the benefits of declarative syntax for ensuring consistent state management, we can now delve deeper into specific practices that promote reproducibility within Ansible scripts.

In IaC, achieving consistent and reliable execution is crucial, particularly for Ansible scripts. This necessitates adherence to several key principles presented below [68]:

- Avoid hard-coding: sensitive information such as credentials and API keys should never be hard-coded within playbooks. Instead, secure management practices like Ansible Vault or environment variables are recommended [11].

- Modular design: promoting code maintainability and facilitating individual component testing can be achieved through a modular design approach, where complex configurations are decomposed into reusable roles [10].

- Dependency management: explicit declaration and management of dependencies within playbooks is crucial to guarantee consistent behavior across diverse environments. Tools like Ansible Galaxy can be employed to effectively manage these dependencies.

- Idempotency: Ensuring idempotency in the scripts are vital. This ensures that scripts produce identical outcomes irrespective of whether they are executed on a pre-existing configuration or not, thereby fostering consistent state management and mitigating the risk of unintended modifications [13].

2.1.3 IaC Platforms

Infrastructure as Code (IaC) platforms provide powerful automation solutions for managing and configuring infrastructure. Among the various options available, tools like Ansible, Puppet, Chef, and Terraform each offer distinct features and approaches to IaC. Some platforms, such as Ansible, are known for their agent-less architecture, which simplifies the setup process and reduces overhead by using protocols like SSH for communication. This approach can be particularly appealing for teams seek-ing a straightforward configuration syntax and ease of use, especially in smaller to medium-sized deployments [68]. On the other hand, tools like Puppet and Chef come with robust capabilities for managing large-scale configurations and employ their own declarative languages, offering extensive control and flexibility. The choice between these platforms often depends on the specific needs of the deployment, including factors like scale, complexity, and the desired ease of adoption.

Table 2.1: Comparison of IaC Tools
Legend: ✗ = Agentless, ✓ = Agent-based, I = Imperative, D = Declarative

Feature	Ansible	Puppet	Terraform
Architecture	✗	✓	✗
Configuration	YAML	DSL	HCL
Syntax	I, D	D	D
Use Cases	Config Mgmt App Deploy Orchestration	Config Mgmt	Infra Provisioning
Strengths	Simple, Flexible Fast Setup	Mature Ecosystem Strong Community	Multi-cloud Immutable Infra

Domain-Specific Language (DSL) is a programming language tailored for specific tasks within a domain, used in tools like Puppet for defining infrastructure in a readable, domain-focused way. HashiCorp Configuration Language (HCL) is a human-readable, machine-friendly configuration language used by Terraform to manage infrastructure across multiple cloud providers with a declarative syntax. YAML is a human-readable data serialization format commonly used for configuration files. It emphasizes simplicity and ease of use, making it popular in tools like Ansible for defining infrastructure as code.

By understanding and implementing these concepts, organizations can leverage Ansible and IaC to achieve greater reproducibility, reliability, and eficiency in managing their IT infrastructure. These practices not only enhance operational stability but also support scalable and agile development processes.

2.1.4 Imperative and Declarative IaC Syntax

IaC can be implemented using either imperative or declarative syntax, each with distinct characteristics and use cases.

Imperative IaC Syntax: This approach involves specifying explicit instructions and steps to provision and configure infrastructure. Developers define the exact sequence of operations needed to achieve the desired state, often using operating system commands. The focus is on control flow and procedural execution [24,60,80]. Ansible, for example, supports an imperative style through yaml-based playbooks where each task specifies particular actions to be executed on the target infrastructure [68].

Declarative IaC Syntax: In contrast, declarative IaC languages describe the target state of the infrastructure without specifying the exact steps to achieve that state. Developers define the desired configuration, and the IaC framework determines the necessary operations to align the infrastructure with the desired state [24, 60, 80]. Puppet is an example of a framework that primarily uses declarative syntax.

Listing 2.1: Ansible code snippet

```
# Imperative Ansible syntax
- name: Check if Apache is installed
  shell: dpkg -l apache2
  register: apache_installed

- name: Install Apache
  shell: apt-get install apache2
  when: apache_installed.rc !=0

# Declarative Ansible syntax
 - name: Install Apache web server
   hosts: webservers
   tasks:
     - name: Install Apache package
       apt:
         name: apache2
         state: present
- name: Start Apache service
  server:
    name: apache2
    state: started
```

Ansible, while capable of declarative syntax, primarily employs an imperative approach. The example above illustrates both styles within Ansible: the imperative code provides step-by-step commands, while the declarative code specifies the desired state.

The choice between imperative and declarative syntax depends on the specific needs of the project and the preferences of the development team. Imperative syntax offers more control over the execution process, while declarative syntax provides a higher level of abstraction and simplicity.

2.1.5 Advanced Concepts in Ansible IaC and Reproducibility

To further enhance our understanding of IaC, particularly with tools like Ansible, it is important to discuss related concepts such as configuration management, infrastructure testing, and environment provisioning. These concepts play a significant role in achieving reproducibility and reliability in infrastructure deployments.

2.1.5.1 Configuration Management

Configuration management involves maintaining the consistency of infrastructure configurations across different environments and over time. Ansible excels in this domain by enabling the definition and automation of configuration tasks through playbooks. This ensures that infrastructure remains consistent and aligned with the desired state, reducing configuration drift and manual intervention [68].

2.1.5.2 Infrastructure Testing

Just as testing is crucial in software development, it is equally important in IaC to ensure that infrastructure configurations work as expected. Tools like Ansible allow for the creation of automated tests to validate infrastructure setups. Testing frameworks such as Testinfra or Molecule can be integrated with Ansible to perform unit and integration tests on infrastructure code, ensuring reliability and functionality [82].

2.1.5.3 Environment Provisioning

Environment provisioning refers to the process of setting up the necessary computing resources and configurations to support application deployments. Ansible facilitates automated environment provisioning by defining infrastructure requirements in code. This allows for rapid and consistent creation of development, testing, and production environments, ensuring that they are identical and reproducible [60].

2.1.5.4 Continuous Integration and Continuous Deployment (CI/CD):

Integrating IaC with CI/CD pipelines enhances the automation of infrastructure changes and deployments. Ansible can be used in conjunction with CI/CD tools such as Jenkins, GitLab CI, or GitHub Actions to automate the testing, provisioning, and deployment of infrastructure. This ensures that changes are tested and deployed in a controlled and repeatable manner, minimizing errors and downtime [28].

2.2 Sustainable AI inference on Edge

2.2.1 Edge Computing

Edge computing refers to the processing of data near the data source, at the edge of the network, rather than relying on a centralized data-processing warehouse. This approach reduces latency, saves bandwidth, and enables real-time processing, which is crucial for applications requiring immediate responses. Edge computing supports various applications, including IoT, autonomous vehicles, and smart cities, by providing local processing power.

By integrating ML and DL models into edge computing, Edge AI allows for advanced analytics and decision-making to occur directly at the data source. This not only enhances performance but also ensures data privacy and security, as sensitive information does not need to be transmitted to central servers.

2.2.2 Edge AI: Machine Learning and Deep Learning at the Edge

The integration of machine learning (ML) and deep learning (DL) models into edge computing environments, referred to as Edge AI, has transformed data processing. Traditionally, ML has enabled systems to learn and adapt from data without explicit programming. Deploying these models directly on edge devices facilitates real-time decision-making and analysis at the data source, leading to applications in predictive maintenance, anomaly detection, personalized healthcare, and smart energy management. However, deploying ML models on edge devices presents unique challenges, including limited computational resources, energy constraints, and the need for lightweight model architectures.

Deep learning, which utilizes sophisticated models capable of uncovering intricate patterns from large datasets, also benefits from deployment on edge devices. This enables real-time data processing and analysis, driving advancements in smart surveillance, intelligent transportation systems, healthcare monitoring, and industrial automation [8, 103]. Similar to ML, the challenges of limited computational resources and energy constraints must be addressed to fully leverage the potential of DL on the edge.

Natural language processing (NLP) is experiencing a shift with the potential deployment of large language models (LLMs) on edge devices. This promises immediate language comprehension and generation, fostering applications in virtual assistants, language translation, and sentiment analysis. However, deploying LLMs on edge devices introduces new challenges, including computational resource limitations, model size constraints, and energy limitations, necessitating the development of streamlined model architectures and optimization techniques.

Fortunately, the development of lightweight AI models, designed for compact and eficient performance, addresses some of these challenges. Techniques such as model pruning, quantization, and knowledge distillation help create compact models that maintain satisfactory accuracy levels. Frameworks like TensorFlow Lite, PyTorch Mobile, and ONNX Runtime further enable the deployment of these models on edge devices, allowing tasks like image classification, object detection, and speech recognition to be performed locally, without relying on constant communication with centralized servers.

2.2.3 Power consumption on Edge

Energy and power measurements play a crucial role in optimizing the performance and eficiency of edge computing systems. By accurately monitoring energy consumption and power usage, ineficiencies can be identified, resource allocation can be optimized, and the battery life of edge devices can be prolonged. Software tools like Perf [6] and NVIDIA-SMI [5], alongside hardware devices such as USB power meters, provide valuable insights into the energy and power consumption of edge devices. This data is instrumental in validating energy-saving techniques and ensuring compliance with energy eficiency standards.

As the demand for AI-driven applications at the edge continues to grow, there is a parallel focus on sustainability and energy eficiency, encapsulated in the concept of GreenAI [40]. GreenAI emphasizes the development and deployment of AI models that are not only powerful but also environmentally sustainable. This is particularly relevant in the context of Edge IoT deployments, where energy consumption is a critical concern due to the limited power availability of edge devices.

The principles of GreenAI involve optimizing model architectures to reduce computational overhead, employing energy-eficient hardware, and leveraging advanced techniques such as federated learning and edge-cloud collaboration. By adopting these practices, organizations can minimize the carbon footprint of their AI deployments while maintaining high performance and accuracy.

Federated learning, for example, allows edge devices to collaboratively train models using local data, reducing the need for extensive data transfer to central servers. This not only enhances data privacy and security but also significantly cuts down on energy consumption associated with data movement. Edge-cloud collaboration, on the other hand, involves distributing computational tasks between edge devices and cloud servers, ensuring that energy-intensive operations are ofloaded to more capable infrastructure while preserving the real-time processing capabilities of edge devices.

2.2.4 Frameworks and Tools for Edge AI

Various frameworks and tools are used in enabling the eficient deployment and management of AI models on edge devices. The most common ones are TensorFlow Lite, PyTorch Mobile, and ONNX Runtime, each offering unique features and capabilities tailored to the needs of edge AI applications.

2.2.4.1 TensorFlow Lite

TensorFlow Lite is a lightweight version of TensorFlow designed for mobile and embedded devices. It provides tools for optimizing models to run eficiently on edge hardware, including support for model quantization and hardware acceleration. TensorFlow Lite's interpreter is designed to work on devices with limited resources, making it ideal for real-time inference tasks on edge devices [44].

2.2.4.2 PyTorch Mobile

PyTorch Mobile extends the popular PyTorch framework to mobile and edge environments. It supports model optimization techniques such as quantization and pruning, allowing developers to deploy eficient models on edge devices. PyTorch Mobile integrates seamlessly with the PyTorch ecosystem, enabling a smooth transition from model development to deployment [77].

2.2.4.3 ONNX Runtime

ONNX Runtime is an open-source project that enables the deployment of models trained in various frameworks, such as TensorFlow and PyTorch, on edge devices. It provides high performance and cross-platform compatibility, making it a versatile choice for edge AI applications. ONNX Runtime supports hardware acceleration and optimization techniques to ensure eficient model execution on resource-constrained devices [73].

These frameworks, combined with advancements in AI hardware and software, are driving the evolution of edge AI, enabling the deployment of sophisticated models that operate eficiently and sustainably in edge environments. By leveraging these tools, developers can create AI-driven solutions that are both powerful and energy-eficient, meeting the demands of modern edge computing applications.

Table 2.2: Comparison of Edge AI frameworks.

Feature	TensorFlow Lite	PyTorch Mobile	ONNX Runtime
Primary Use Case	Mobile and Embedded Devices	Mobile and Edge Devices	Cross-Platform Model Deployment
Model Optimization	Quantization, Pruning	Quantization, Pruning	Quantization, Hardware Acceleration
Supported Languages	Python, Java, Swift, C++	Python	Python, C++
Hardware Acceleration	Yes, via delegates (e.g., NNAPI, GPU)	Yes (e.g., NNAPI, Metal)	Yes (e.g., TensorRT, OpenVINO)
Model Format	TFLite	TorchScript	ONNX
Compatibility	TensorFlow Models	PyTorch Models	Models from various frameworks (TensorFlow, PyTorch, etc.)
Ease of Use	High, with extensive documentation	High, integrates with PyTorch	Moderate, requires conversion to ONNX format
Performance	Optimized for mobile and embedded	Optimized for mobile and edge	High performance with cross-platform support
Community Support	Large, with many resources	Large, integrated with PyTorch community	Growing, with broad framework support
Deployment Flexibility	Primarily for Android and iOS	Primarily for Android and iOS	Cross-platform (Windows, Linux, macOS, Android, iOS)

Chapter 3

Related Works

3.1 IaC Reproducibility

This section provides an in-depth overview of various research studies that have investigated IaC tools, techniques, and practices. It also summarizes existing work on reproducibility practices in software engineering, positioning our study within the broader context of the field.

3.1.1 Assuring IaC Script Quality

To ensure the quality of IaC scripts, we categorize the related work into different subfields that focus on best practices, code smells, and security aspects.

3.1.1.1 Best Practices in IaC

Several studies have explored tools and techniques to ensure various quality aspects of IaC, including security and maintainability.

Kumara et al. [60] addresses the growing significance of infrastructure-as-code (IaC) within the DevOps paradigm, highlighting the need for speed in software development and deployment. Despite its widespread adoption, the academic literature on IaC remains limited, particularly concerning its maintenance and evolution. To bridge this gap, the authors conducted a systematic review of gray literature, analyzing 67 high-quality sources to uncover best and bad practices in IaC development across various languages, including Puppet, Chef, and Ansible. They proposed a rigorous definition of infrastructure code and established a taxonomy categorizing ten best practices and four bad practices, emphasizing the importance of implementation, design, and adherence to core IaC principles. The findings reveal both significant challenges—such as conflicting best practices and security issues—and valuable insights into the most commonly used IaC languages and practices. The study concludes with a call for further research in the area of IaC maintenance and security, outlining plans for future work that aims to automate recommendations for best practices and expand the investigation to encompass a broader range of IaC tools.

Rahman et al. [80] focuses on the significant impact of defects in infrastructure-as-code (IaC) scripts, which can lead to major system outages. The authors create a detailed defect taxonomy by analyzing 1, 448 defect-related commits from OpenStack's open-source software repositories. To ensure the taxonomy's relevance, they survey 66 practitioners to gauge their agreement with the identified defect categories. The resulting taxonomy includes eight categories, with a specific emphasis on idempotency, which highlights defects that affect system provisioning during multiple executions of the same IaC script. The analysis identifies configuration data as the most frequent defect category, reflecting issues with erroneous configuration inputs. By quantifying these defects across 80, 425 commits from 291 repositories, the authors aim to improve the understanding and overall quality of IaC scripts, offering valuable insights for practitioners in enhancing their infrastructure code practices.

Hummer et al. [49] present a model-based testing approach for Infrastructure as Code (IaC) aimed at verifying the idempotence of automation scripts, such as those written in Chef. The authors highlight the critical importance of idempotence for ensuring that IaC automation can consistently bring systems to a desired state, regardless of their starting conditions. Through an extensive evaluation of approximately 300 real-world Chef scripts, the proposed framework successfully identified nearly one-third as non-idempotent, also uncovering a bug in the Chef implementation. The study emphasizes the need for systematic testing in the context of IaC, paving the way for future research that could extend the approach to distributed automation, other IaC frameworks like Puppet, and the identification of implicit dependencies. This work contributes significantly to the understanding and improvement of IaC script quality, highlighting the ongoing challenges in ensuring robust automation practices.

3.1.2 Code Smells in IaC Scripts

Research on code smells in IaC scripts has been extensive, focusing on identifying and mitigating issues that affect script quality.

Dallapalma et al. [30] introduces a catalog of 46 software quality metrics specifically designed for Infrastructure as Code (IaC), with a focus on Ansible. As IaC practices continue to gain traction in the industry, the authors emphasize the need for measurable approaches to maintain and improve code quality. The proposed metrics aim to aid DevOps engineers in assessing infrastructure code properties, identifying potential defects, and facilitating incremental refactoring. By providing a structured set of metrics tailored to the unique characteristics of IaC scripts, this work lays the groundwork for future empirical studies on the relationship between these metrics and code quality. The authors also highlight the importance of generalizing their findings to other IaC frameworks, paving the way for a comprehensive understanding of IaC quality across different languages and tools.

Schwarz et al. [87] addresses the challenge of maintaining high quality in Infrastructure as Code (IaC) by investigating code smells, a concept borrowed from traditional software engineering. While prior research has applied code smells to Puppet, this study extends the analysis to Chef, examining both open and closed source IaC repositories through two case studies. The findings reveal that IaC smells are prevalent across different technologies and can be defined in a technology-agnostic manner. Additionally, the authors introduce 17 new code smells that have not been previously explored in the IaC domain. This work contributes to the ongoing discourse on IaC quality by providing a comprehensive catalog of code smells that can help assess and improve IaC practices.

Sharma et al. [89] presents a quality analysis of Infrastructure as Code (IaC), specifically focusing on Puppet configuration code. The authors propose a catalog of 24 configuration smells, 13 implementation and 11 design smells, derived from established best practices. Analyzing 4,621 Puppet repositories with over 8.9 million lines of code, the study addresses key research questions regarding the distribution, co-occurrence, and density of these smells. Findings indicate that design configuration smells tend to co-occur more frequently and exhibit negative correlation with project size, highlighting the need for careful design decisions. The study not only contributes to the understanding of configuration code quality but also provides practical tools for practitioners to identify and mitigate configuration smells, thereby promoting maintainability in IaC practices.

Rahman et al. [83] investigates the correlation between specific source code properties and defects in Infrastructure as Code (IaC) scripts, emphasizing the impact of these defects on the reliability of automated deployment pipelines. Through qualitative analysis of defect-related commits from open-source repositories, the authors identify ten source code properties associated with defective IaC scripts, with lines of code and hard-coded strings showing the strongest correlations. A survey of practitioners reveals significant agreement on the relevance of executing external modules and the use of hard-coded strings. The study further develops defect prediction models based on these properties, achieving precision scores between 0.70 and 0.78, and recall rates of 0.54 to 0.67. The findings suggest that practitioners should focus their inspection and testing efforts on IaC scripts exhibiting these identified properties to enhance code quality.

3.1.3 Security Smells in IaC Scripts

Detecting security smells in IaC scripts has been a significant area of research.

Dai et al. [29] presents SecureCode, an analysis framework designed to automatically extract and assess embedded scripts within infrastructure code, specifically targeting risky patterns that can adversely affect the entire infrastructure. By focusing on Ansible playbooks, the framework detects these risky patterns along with their correlated severity levels and potential negative impacts. Integrated into a DevOps pipeline on IBM Cloud, SecureCode was tested on 45 IBM Services community repositories, successfully identifying 3,419 true issues with 116 false positives within a short time frame. Notably, 1,691 of these identified issues were classified as having high severity levels, highlighting the tool's effectiveness in enhancing infrastructure code security.

Opdebeeck et al. [75] investigates the challenges associated with variable management in Ansible, a widely used Infrastructure as Code (IaC) language. It identifies six novel code smells stemming from Ansible's complex variable precedence rules and lazy-evaluated template expressions, which can lead to significant infrastructure defects. Utilizing a transposed program dependence graph for accurate control and data flow representation, the authors detect these smells in 21,931 open-source Ansible roles, uncovering 31,334 unique instances. The findings reveal an increasing trend in variable smells over time, with changes often introducing new smells rather than resolving existing ones. This research emphasizes the need for enhanced quality checkers in IaC and advocates for a deeper understanding of the semantics beyond mere syntax in IaC practices.

They further extended their work by introducing GASEL [76], a novel security smell detector for Ansible, addressing the limitations of previous static analyses that overlook control and data flow in Infrastructure as Code (IaC) scripts. GASEL employs graph queries on program dependence graphs to identify seven security smells, improving both precision and recall compared to existing detectors. The evaluation against an oracle of 243 real-world security smells demonstrates the effectiveness of this approach. Additionally, the study reveals that over 55% of security smells exhibit data-flow indirection, and more than 32% necessitate whole-project analysis for detection. These insights underscore the need for more sophisticated static analysis tools to effectively identify security vulnerabilities in IaC.

Bhuiyan et al. [18] addresses the problem of insecure coding patterns (icp) in Infrastructure as Code (IaC) scripts, which can create vulnerabilities in automated deployment pipelines. The study focuses on characterizing co-located icp—patterns that occur together in a script—and aims to help practitioners prioritize their code review efforts. By analyzing 7,222 Puppet scripts from Mozilla, Openstack, and Wikimedia, the authors identify frequently co-located icp using association rule mining. They find that 21.06% of the scripts contain co-located icp, with hard-coded secrets and suspicious comments being the most common. The study concludes that prioritizing code reviews for scripts with co-located icp can improve the detection and mitigation of security weaknesses in IaC scripts.

3.1.4 Reproducibility Practices in Software Engineering

Reproducibility of scientific experiments and software engineering practices is a critical concern addressed in various studies.

Gonzalo et al. [85] highlighted the importance of good coding practices, consistent coding styles, documentation, and error handling for reproducibility. They advocated for the use of version control, comprehensive documentation of code dependencies, simplified execution processes, and containerization technologies such as

Docker.

Feitelson et al. [36] suggested that thorough project documentation is essential for enhancing the reproducibility and comparability of scientific experiments, emphasizing the need for clear and detailed records of experimental setups and procedures.

3.1.5 Comparison with Existing Work

Previous research in the IaC domain has predominantly focused on examining security and maintainability aspects within IaC scripts. However, our work brings attention to the often-overlooked yet vital concept of reproducibility. While security and maintainability are crucial quality attributes, ensuring the reproducibility of IaC deployments is equally important for achieving dependable and consistent infrastructure management.

Our research makes significant contributions by first creating a comprehensive catalog of reproducibility smells through an aggregation of relevant information from diverse sources. We then develop a reproducibility smell detector for Ansible scripts, named Reduse. Using this tool, we conduct an empirical study to explore the prevalence of reproducibility smells in open-source repositories and analyze the relationships among these smells. This investigation provides deeper insights into reproducibility issues in the IaC domain and contributes to the improvement of IaC script quality and overall system resilience.

3.2 Sustainable AI inference on Edge Related Works

Machine learning (ML) and deep learning (DL) on edge devices have emerged as powerful technologies for enabling real-time AI applications at the network's periphery. However, deploying complex AI models on resource-constrained edge devices presents significant challenges. This section reviews related research on deep learning for edge computing, focusing on performance analysis, optimization techniques, and potential applications.

Table 3.1: Summary of Related Works and Comparison with Current Research

Study	Focus	Key Contributions	Metrics/Methods
Kumara et al. [60]	Best Practices in IaC	Systematic review of best and bad practices in IaC	Analysis of 67 sources
Rahman et al. [80]	Defect Taxonomy in IaC	Defect taxonomy and analysis of OpenStack commits	Analysis of 1,448 defect-related commits
Hummer et al. [49]	Automated Testing in IaC	Model-based testing for idempotence in Chef scripts	Evaluation of 300 real-world scripts
Dallapalma et al. [30]	Quality Metrics for IaC	Catalog of 46 quality metrics for Ansible	Metric cataloging and analysis
Schwarz et al. [87]	Code Smells in IaC	Catalog of 17 new code smells for Chef	Analysis of IaC repositories
Sharma et al. [89]	Configuration Smells in Puppet	Catalog of 24 configuration smells	Analysis of 4,621 Puppet repositories
Rahman et al. [83]	Automated Detection of Code Smells	Development of AnsibleCheck framework	Qualitative analysis and defect prediction models
Dai et al. [29]	Security Smells in IaC	SecureCode framework for risky patterns	Analysis of Ansible playbooks
Opdebeeck et al. [75]	Variable-related Smells in Ansible	Detection of 6 novel variable-related code smells	Analysis of 21,931 Ansible roles
Opdebeek [76]	Security Smells in Ansible	GASEL detector for security smells	Graph-based analysis of program dependence graphs
Bhuiyan et al. [18]	Insecure Coding Patterns	Prioritization of co-located insecure coding patterns	Analysis of 7,222 Puppet scripts
Gonzalo et al. [85]	Reproducibility Practices	Importance of coding practices, documentation, and containerization	Advocacy for version control and Docker
Feitelson et al. [36]	Documentation for Reproducibility	Importance of detailed project documentation	Review of project documentation practices
Our Work	Reproducibility of IaC Scripts	Identification of reproducibility smells and tool development	Detection tool for reproducibility smells, analysis of Ansible playbooks

3.2.1 Performance Analysis and Hardware Platforms

Various studies have investigated the performance of different deep learning mod-els on multiple edge hardware platforms. Rafal et al. [79] benchmarked inference times for popular convolutional neural network (CNN) architectures like MobileNet, EficientNet, VGG, ResNet, and InceptionV3 on edge platforms such as NVIDIA Jetson Nano, Intel Neural Stick, and Google Coral USB Dongle Their findings indicate that Google platforms offer the fastest average inference times, particularly for newer models like MobileNet and EficientNet, while the Intel Neural Stick stands out for its versatility in running a broader range of architectures.

3.2.2 Software Frameworks

The choice of software framework significantly impacts the eficiency of deep learning on edge devices. Zhang et al. [107] compared popular frameworks such as TensorFlow, Caffe2, MXNet, PyTorch, and TensorFlow Lite across metrics like latency, memory footprint, and energy consumption. Their study reveals that TensorFlow performs well for large-scale models on CPU-based platforms, Caffe2 excels with small-scale models, PyTorch demonstrates memory eficiency, and MXNet proves to be energy-eficient on specific devices.

3.2.3 Energy and Runtime Eficiency

Energy consumption and runtime performance are crucial considerations for edge deployments. Georgiou et al. [40] compared TensorFlow and PyTorch for training and inference of recommender systems, computer vision, and natural language processing models. Their findings suggest that TensorFlow is generally more eficient for recommender systems and ResNet-50 models during training, whereas PyTorch shows superior energy eficiency and runtime performance across other model types, particularly for NLP tasks during the inference phase.

3.2.4 Model Compression and Deployment

Deploying large DL models on edge devices often requires model compression techniques to reduce their size and memory footprint.

Rahman et al. [84] showcased the effectiveness of quantization techniques in achieving comparable accuracy to original models while enabling deployment on resource-constrained devices. Their study focused on fine-tuning MobileBERT, a compact version of BERT, for reputation analysis using TensorFlow Lite models and quantization for optimization. The results demonstrate significant reductions in model size and faster inference times with minimal accuracy loss.

Frameworks such as TensorFlow Lite and Microsoft's EdgeML [8] facilitate the deployment of compressed neural networks on various IoT devices. These advancements enable a wide range of TinyML applications, including environmental monitoring, sign language detection, and medical face mask detection, leveraging real-time analysis and interpretation of data at the edge.

3.2.5 Challenges and Optimization Techniques for Deep Learning on Edge Devices

While deep learning offers exciting possibilities for edge computing, resource limitations and the need for eficient model execution pose significant challenges.

Qi et al. [78] explore these challenges and discuss three key optimization techniques: parallel acceleration, quantization, and model pruning. Their study emphasizes the importance of a holistic approach that considers both hardware and machine learning algorithm levels for successful deep learning deployment on edge devices. They highlight the need for balancing precision reduction during model quantization with hardware-specific adaptations to achieve optimal performance.

Similarly, Ghosh et al. [41] address the challenges of implementing deep learning on resource-constrained IoT devices. They propose a hybrid approach that combines edge and cloud computing for IoT data analytics. Their study leverages autoencoders, a type of neural network for dimensionality reduction, by deploying the encoder part on the edge to reduce data size before sending it to the cloud for further processing. This approach significantly reduces data transmission without compromising the accuracy of ML tasks, as demonstrated in their evaluation of human activity recognition from smartphone data.

3.2.6 Applications of Deep Learning on Edge Devices

The intersection of deep learning and edge computing offers numerous advantages for real-world applications.

Wang et al. [103] explore this synergy, highlighting the need for powerful deep learning capabilities to handle complex scenarios like video analytics at the edge of IoT networks. They discuss how edge computing provides purpose-built hard-ware platforms, such as the lightweight Nvidia Jetson TX2 developer kit, to sup-port deep learning tasks at the network edge. Their study delves into various deep learning models applicable to edge computing, including restricted Boltzmann machines (RBMs), auto-encoders (AEs), deep neural networks (DNNs), convolutional neural networks (CNNs), recurrent neural networks (RNNs), and deep reinforcement learning (DRL). They identify key challenges such as model training, inference optimization, application enhancement, and hardware-software co-design. Finally, they explore real-world applications across various domains, including smart multimedia, transportation, cities, and industries, emphasizing the benefits of edge computing in terms of low latency, energy savings, context-aware services, and privacy.

3.2.7 Comparison with Existing Work

While prior research has made significant contributions to deploying machine learning models on edge devices, our work offers several key advancements that extend the current state-of-the-art:

- Breadth of Model Selection: Existing studies often focus on a limited set of models or specific categories, such as CNNs for computer vision tasks [79]. Our work, in contrast, investigates a broader variety of models across multiple categories, including large language models (LLMs). This new category opens doors to novel applications on the edge, such as voice assistants, demonstrating the applicability of our approach to a diverse range of edge computing tasks.

- Deeper Performance Analysis: Previous comparisons primarily focus on model accuracy between original and compressed versions [84, 103]. We extend this analysis by measuring critical performance metrics beyond accuracy, incorporating inference time, memory utilization, and power consumption measurements using hardware devices. This comprehensive evaluation provides a more holistic understanding of the trade-offs between model complexity and resource usage on edge devices, offering a more nuanced perspective on model selection and optimization for edge deployment.

- Power Consumption and Guidelines: While some studies acknowledge energy eficiency as a concern [40,84], they often lack in-depth analysis or concrete recommendations. Our work goes beyond simply acknowledging the importance of power consumption by directly measuring power consumption using hardware devices. We leverage these insights to develop practical guidelines for optimizing model selection and deployment strategies for power-constrained edge environments.

By addressing these limitations, our work offers a more comprehensive and practical approach to deploying deep learning models on edge devices. We consider a wider range of models, delve deeper into performance analysis, and provide practical guidance for power-eficient model selection and deployment on resource-constrained edge hardware.

Table 3.2: Summary of Related Works on Sustainable AI Inference on Edge Devices

Study	Focus	Key Contributions	Metrics/Methods
Rafal et al. [79]	Performance Analysis	Benchmarking of CNN architectures on edge platforms	Inference times for MobileNet, EfficientNet, VGG, ResNet, and InceptionV3 on NVIDIA Jetson Nano, Intel Neural Stick, Google Coral USB Dongle and PCIe
Zhang et al. [107]	Software Frameworks	Comparison of frameworks for deep learning eficiency	Metrics: latency, memory footprint, energy consumption for TensorFlow, Caffe2, MXNet, PyTorch, TensorFlow Lite
Georgiou et al. [40]	Energy and Runtime Eficiency	Comparison of TensorFlow and PyTorch for various tasks	Energy eficiency and runtime performance for recommender systems, computer vision, and NLP models
Rahman et al. [84]	Model Compression	Effectiveness of quantization techniques for MobileBERT	Model size reduction, inference times, accuracy loss for quantized MobileBERT models
mi et al. [8]	Model Deployment	Deployment of compressed neural networks using TensorFlow Lite and EdgeML	Application in TinyML for environmental monitoring, sign language detection, medical face mask detection
Qi et al. [78]	Optimization Techniques	Optimization techniques for deep learning on edge devices	Techniques: parallel acceleration, quantization, model pruning
Ghosh et al. [41]	Edge and Cloud Hybrid Approach	Hybrid edge-cloud approach for IoT data analytics	Use of auto-encoders for dimensionality reduction, data transmission optimization
Wang et al. [103]	Applications of Deep Learning	Exploration of deep learning models for edge computing	Models: RBMs, AEs, DNNs, CNNs, RNNs, DRL; Applications in multimedia, transportation, smart cities
Our Work	Sustainable AI Inference	Advancements in model selection, performance analysis, and power consumption	Evaluation of model complexity, inference time, memory utilization, power consumption

Chapter 4

Reproducibility practices in Infrastructure as Code

4.1 Study Design

This section presents our comprehensive approach to examining the reproducibility of Infrastructure as Code (IaC) and the sustainability of AI inference on edge devices. We aim to uncover challenges and propose solutions in both areas, ensuring robust, reproducible IaC practices and energy-eficient AI deployments on edge devices.

4.1.1 Study Overview

The main objective of this part is to understand the challenges involved in reproducing computing infrastructure within the context of Infrastructure as Code (IaC). Specifically, we aim to identify and analyze the programming practices that lead to these reproducibility challenges, which we refer to as reproducibility smells. Furthermore, we seek to develop automated methods for detecting these reproducibility issues to investigate their prevalence in open-source Ansible projects. By doing so, we hope to gain a better understanding of the relationships among these smells and ultimately guide developers in creating more reproducible IaC scripts. To achieve these goals, we address the following research questions (RQs):

RQ1. What kind of programming practices impact reproducibility in IaC scripts?
Identifying practices that impede reproducibility allows us to provide guidance to developers for creating scripts that consistently produce the desired infrastructure setups across different environments. This research question aims to consolidate the scattered knowledge about reproducibility in the IaC domain and create a comprehensive understanding of the factors affecting reproducibility.

RQ2. Which reproducibility smells are more prominent in open-source repositories?
This research question seeks to determine if certain reproducibility smells are

more common than others in analyzed open-source software repositories. Understanding the prominence of specific smells can help developers be more vigilant about those that are likely to occur frequently, thereby encouraging them to take appropriate measures to address these issues.

RQ3. Do reproducibility smells co-occur?
Exploring the co-occurrence of different types of smells can provide valuable insights into their occurrence patterns and inter-dependencies. Identifying correlations between smells can reveal hidden complexities in IaC scripts, which can aid developers in adopting comprehensive approaches to improve script quality and reproducibility.

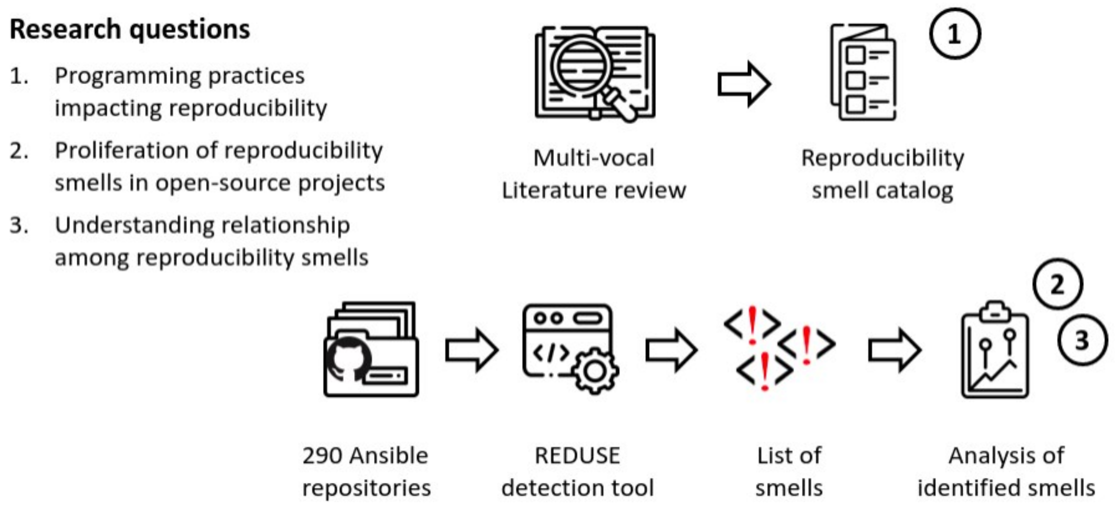

Research questions
1. Programming practices impacting reproducibility
2. Proliferation of reproducibility smells in open-source projects
3. Understanding relationship among reproducibility smells
Multi-vocal Literature review
Reproducibility smell catalog
1
2
3
290 Ansible repositories
REDUSE detection tool
List of smells
Analysis of identified smells

Figure 4.1: Overview of the IaC Reproducibility study.

Our study design consists of several interconnected steps aimed at answering the aforementioned research questions, as illustrated in Figure 4.1. We outline these steps below:

4.1.1.1 Multi-Vocal Literature Review

To address RQ1, we conduct a multi-vocal literature review. This involves searching, filtering, and consolidating existing literature to identify programming practices that impact reproducibility. The outcome of this review is the creation of a comprehensive catalog of reproducibility smells specific to the IaC domain. This catalog serves as the foundation for our further analyses and tool development.

4.1.1.2 Tool Development for Detecting Reproducibility Smells

Based on the identified reproducibility smells, we develop a tool, Reduse, designed to detect these smells in Ansible scripts. This tool automates the detection process, enabling us to eficiently analyze large volumes of IaC code for reproducibility issues.

4.1.1.3 Empirical Analysis of Open-Source Ansible Repositories

To answer RQ2 and RQ3, we curate a set of top open-source Ansible repositories from the Ansible Galaxy platform [1]. Using Reduse, we detect reproducibility smells in these repositories. This empirical analysis allows us to investigate the prevalence of different reproducibility smells and their co-occurrence patterns.

4.1.1.3.1 Data Collection and Analysis We systematically collect data on the detected reproducibility smells from the analyzed repositories. This data forms the basis for our analysis, where we examine the frequency and distribution of different smells. We also analyze the relationships among smells to understand their inter-dependencies and potential clustering patterns.

4.1.1.3.2 Observations and Insights Finally, we provide our observations and insights based on the analysis of the collected data. These findings help us to better understand the challenges associated with reproducibility in IaC scripts and offer practical recommendations for developers to mitigate these issues. Our study not only highlights the importance of reproducibility in IaC but also contributes valuable tools and knowledge to the field, enhancing the overall quality and reliability of IaC practices.

4.2 Research Methodology

This section elaborates our approach to create a catalog of reproducibility smells and describes them using a template.

4.2.1 Catalog Creation

Given the practical nature of the problem space, we carry out a Multi-vocal Literature Review (mlr) [39]. An mlr combines information in academic research papers with gray literature sources, including blog posts, articles, programming discussion forums, and oficial documentation provided by framework or library developers. A wide range of resources helps us gain insights into the diverse range of efforts related to good and bad IaC practices. With this extensive literature search and catalog creation effort, we contribute to the state of the art by thoroughly searching the academic and gray literature for reproducibility issues, systematically prune and group these issues into distinct, actionable smells, and clearly define and catalog them. Figure 4.2 summarizes our adopted process to conduct the mlr. We elaborate on the adopted process in the rest of the section.

4.2.1.1 Resource gathering and searching

We follow practices employed in multi-vocal literature review [39,56,94,101] to search, filter, and identify resources for the review. Specifically, we use the Google search engine to search the gray literature, focusing on text-based sources such as reports, blog posts, white-papers, and oficial documentation related to IaC platforms. In the case of academic literature, we consider Google Scholar as well as ieee and acm digital libraries for literature search.

We used these resources as our primary sources for literature due to their direct access to full-text, peer-reviewed content, which is essential for in-depth analysis and citation accuracy. ACM and IEEE are renowned for their high-quality, rigorously vetted publications in computer science and engineering, making them industry standards. Google Scholar, with its broad coverage and citation tools, allowed us to perform comprehensive searches across a wide range of disciplines. Apart from the considered digital libraries and search engine, there are a few bibliographic indexing services such as DBLP. Given that such indexing services do not hold full-text, we adopted ACM, IEEE, and Google Scholar as our literature search sources.

We formulate the following search queries, inspired by similar studies [24, 61, 86], considering the scope of our search:

'Infrastructure as code' + smells + reproducibility

‘Infrastructure as code’ + anti-patterns + reproducibility

‘Infrastructure as code’ + bugs + reproducibility

‘Ansible best practices’ + reproducibility

‘Ansible bad practices’ + reproducibility

‘Ansible anti-patterns’ + reproducibility

We carefully examine the search results corresponding to each search string and document the relevant resources (by reading the title of the resources) until we do not find new resources.

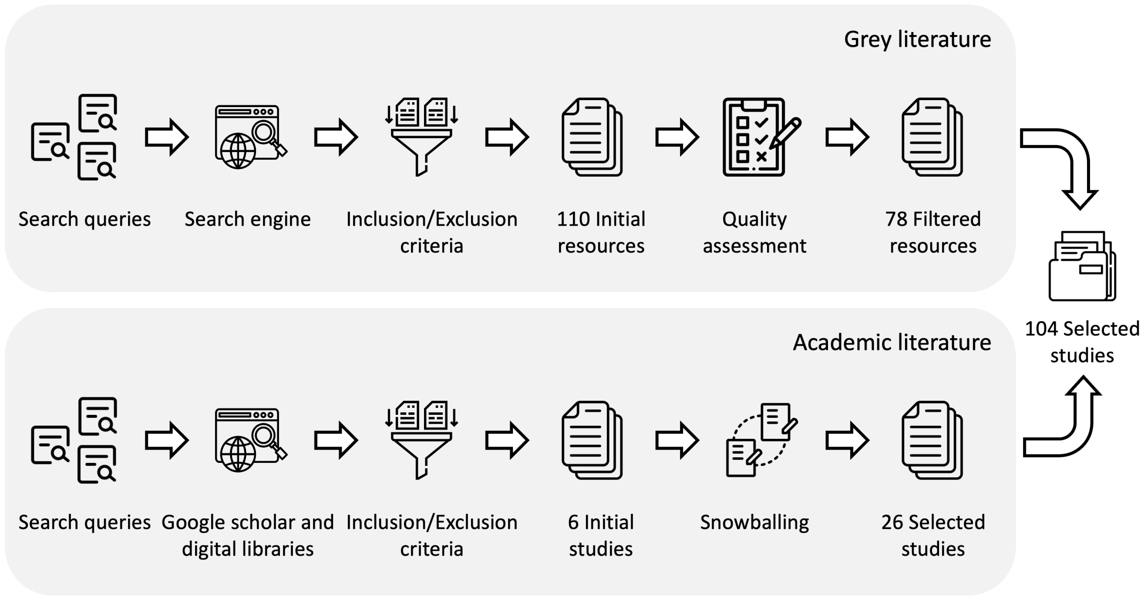


Figure 4.2: Overview of the multi-vocal literature review process.

In summery, we got 45, 40, 22, 15, 23, 18 documents for each of the search queries, respectively. In total we got 163 grey literature documents after applying inclusion and exclusion criteria. We also did an additional round of quality assessment on the documents which got us 78 documents.

Our replication package [9] includes the metadata of the resources (such as url and title) along with the number of resources collected from individual search queries.

4.2.1.2 Inclusion and exclusion criteria

We define inclusion and exclusion criteria to filter out irrelevant resources. The inclusion criteria are as follows.

- The article must be written in English and have accessible full text.
- The article should align with the focus of this study i.e., covering practices to follow or avoid, discussing bugs, defects, smells, or anti-patterns, or describing challenges related to reproducibility in IaC, in general, or Ansible as a specific IaC automation framework.

We exclude resources from our analysis that meet at least one of the following criteria.

- Duplicate articles found across various sources.
- Short articles such as extended abstracts.
- Articles that do not discuss reproducibility aspects, irrespective of whether IaC or not.
- Articles that do not provide an adequate scope, rationale, consequences, or examples of recommended practices or practices to avoid such as bugs, defects, smells, and anti-patterns.

As Figure 4.2 shows, we obtain a list of six academic and 110 gray literature resources after applying inclusion and exclusion criteria.

4.2.1.3 Snowballing

We employ both backward and forward recursive snowballing by carefully reviewing all the references and citations of our primary academic articles to identify relevant resources. In backward snowballing, we use the references of primary academic articles, while in forward snowballing, we explore citations for additional materials. Each potentially relevant article that we find through snowballing also goes through our inclusion and exclusion criteria. We apply the snowballing process to all the references and citations for the identified academic studies. when identified papers no longer align with our specified research interests such as IaC best practices, maintaining focus and ensuring that the selected literature makes meaningful contributions to the exploration of IaC best practices. For each potentially relevant article that we identify through snowballing, we then apply our inclusion and exclusion criteria. We add a list of 26 articles from this exercise.

4.2.1.4 Quality assessment of grey literature

To ensure a robust literature review, we have established a systematic process for assessing academic literature. This quality assessment involves evaluating each paper's alignment with our research focus on IaC principles, using discerning criteria to assess relevance and significance. Integrating gray literature sources can provide untapped information in pursuing comprehensive coverage of the knowledge in the field. However, the nature of gray literature necessitates a careful evaluation to ensure the credibility and relevance of these sources. To this end, We follow the quality assessment best practices suggested in literature [24, 38, 61].

We evaluate each gray resource on a scale of 20 points, covering reputation of the publishing venue, author's expertise, clarity of its purpose, and publishing date. Given that existing literature do not provide a systematic guidelines to evaluate and assign a score to gray resources beyond the above-mentioned evaluation criteria, we apply the following mechanism to thoroughly examine each resource.

- To assess venue reputation, a weighted approach considers afiliations (i.e., institutional ties and collaborations), endorsements (i.e., organizational support), publication history (including publication frequency and citation impact) of the venue, and reviews (in the form of user feedback). All of these parameters had the same weight.

- In evaluating author expertise, we consider academic credentials (i.e., author's educational and professional background as well as specialization), publication track record, research afiliations, and professional experience.

- The evaluation of content clarity includes factors such as relevance, clarity, thorough analysis, novel insights, and illustrative examples.

- Publication date is used as a metric of recentness. Recently released resource scores higher than a relatively old resource.

By employing these criteria, reviewers make informed decisions about incorporating reliable, relevant, and recent scholarly contributions into their assessments.

The quality assessment process involves a thorough evaluation conducted by two reviewers; both are graduate students with four years of experience in software development. One of the reviewers possesses expertise in the IaC field, whereas the second has a general acquaintance. Each reviewer independently examined the resources by accessing the links and assessed them according to above-mentioned criteria. Each reviewer assign a score corresponding to each of the criterion mentioned above with 5 being the highest score and 1 is the lowest. All four aspects have the same weight in calculating the final score. After completing the exercise independently, we consolidate the scores. We obtain a high inter-rater agreement ($\kappa = 0.94$). If their individual scores for a specific aspect have a minor disagreement (i.e., ± 1), the consolidated score is calculated by taking an average. However, when their individual scores have a substantial disparity, both reviewers engage in a discussion to understand the rationale for the provided scores and reach a consensus on the final score.

After evaluating all the resources, we discard resources with a score of less than 10 in the quality assessment exercise. We also discard resources with a score less than three (out of five) in the Relevance and clarity of content aspect to keep the selected resources very relevant to our study, resulting in a total of 78 resources remaining. The marking guideline, the assessment document of each reviewer, and the detailed evaluation scores corresponding to each gray resource, including a summary, and the resource link, can be found in our replication package [9].

4.2.1.5 Data extraction and analysis

We combine resources obtained from both the academic and gray literature selection and filtering processes. We thoroughly study the combined list of resources and document their summary, key learning, and relevant metadata (e.g., BibTex entry). We use the information summarized from the selected studies and resources to create a catalog of reproducibility smells.

We employed open coding [59], a qualitative method for analyzing and organizing concepts, to systematically categorize information from summaries of research papers and code examples. The lead author, with prior experience in qualitative research, conducted the initial coding phase. This involved thoroughly reviewing the collected information, breaking it down into smaller segments for detailed examination, and identifying relationships, similarities, and differences. Each segment received a descriptive label reflecting the main ideas or practices related to reproducibility. To ensure consistency and capture emerging themes, an iterative refinement approach was adopted.

The lead author revisited segments and codes multiple times, and after each round of coding, other authors reviewed the codes and participated in further rounds of coding. This collaborative process continued until a consensus was reached on a final coding scheme that effectively captured the data. Employing emergent coding throughout, we ultimately identified and categorized six key concepts, which we then classified as reproducibility smells. Through this process we were able to categorize the issues listed in the different documents as a smell catalog.

4.2.2 Results of RQ1—Reproducibility Smells Catalog

A reproducibility smell is a practice to specify, configure, or program IaC scripts that hinder the reproducibility of the script. Each smell captures a concrete practice violating IaC principles and best practices leading to compromised reproducibility. These smells often arise from practices that directly embed sensitive information within the script, rely on external dependencies not managed by the script itself, or lack proper version control. Such practices can lead to inconsistencies when running the script in different environments or at different times.

Considering that code smells inherently signal a potential issue rather than an actual problem [90], reproducibility smells similarly point to a violation of guidelines and a potential issue necessitating additional validation within the given context.

We describe each smell in our catalog using a name, description, and example following potential fixes. In addition each smell is followed by the detection rules that is used to detect the smell. We keep the code snippets for the discussed examples in our replication package [9]. Similarly, we include the most important references for each proposed reproducibility smell; an interested reader may find the full list of references, including pointing to gray literature, in our replication package.

4.2.2.1 Broken dependency chain

A broken dependency chain occurs when a dependency or a required component, such as a package, cannot be installed or configured, which prevents the script from executing successfully [50, 54, 60, 71]. This disrupt system reproducibility by causing inconsistent environments across deployments. This inconsistency undermines the ability to reliably recreate identical system states, crucial for predictable outcomes in software development and deployment. Effective dependency management is essential for ensuring reproducible builds and deployments.

Example: In an Ansible playbook belonging to ansible-jupyterhub-hpc repository, a task 'install CHP proxy auth token', gets tokens from a source file, then uses these tokens to perform several operations [22]. However, the task does not check the existence of the source file nor the correctness of the extracted tokens, which can lead to a broken dependency chain.

Potential fix: In this example, adding Ansible constructs stat and when that check the existence of the file and the correctness of the tokens can mitigate the issue [13].

Detection mechanism: We detect this smell in an Ansible task when the task installs a package using hard-coded keys, uses fixed authorization tokens, or installs a package or library directly from a fixed URL or repository. In addition, missing files in specified paths can lead to failed installation or a misconfigured library or package. If any of the following conditions are true, we detect this smell in an Ansible task.

- The task involves installing packages using hard-coded keys or tokens.
- The task installs packages using a hard-coded url.
- The task installs packages using an invalid url.
- The task uses a set of files but does not check the file's existence.

We do not detect the smell when the task requires to use of at least one of the checking constructs to ensure the correct execution of the task: package-facts, debug,when, set-fact, assert, with-items and set-facts [31, 54, 71].

4.2.2.2 Outdated dependency

This smell occurs when an Ansible script specifies an outdated version of a software library or package for installation [31,35,54,63,70,104]. Outdated dependencies in Ansible scripts can lead to inconsistencies across environments due to version variations, and expose the infrastructure to hidden bugs, regressions, and security vulnerabilities, ultimately hindering script reproducibility.

Using an old version of a library can significantly impact the reproducibility of IaC scripts. Older versions may cause dependency conflicts, contain security vulnerabilities, and rely on deprecated features, all of which can lead to errors and inconsistent behavior across different environments. Additionally, updates in newer versions often include bug fixes, performance improvements, and behavioral changes that enhance compatibility and functionality. Limited support and outdated documentation for older versions further hinder troubleshooting and understanding. Therefore, to ensure reliable and consistent reproduction of results, it is crucial to use up-to-date library versions, which also benefit from improved security, compatibility, and community support.

Example: An Ansible playbook [32] in repository `ansible-role-virtualenv` has a task named '`install virtualenv post packages`'. It installs the packages that are specified in a default list specified within the virtual environment package manager which could become outdated because no update policy is mentioned in the script and the version in the package manager is fixed.

Potential fix: It is a common practice to create a requirement file with the compatible versions. Similarly, creating Docker images with all the necessary packages and libraries is also a typical practice. However, updating the dependencies and their corresponding stable versions is strongly recommended to avoid this smell. In the above example, we add a `state` Ansible property that only installs the missing packages with their latest stable and compatible version.

Detection mechanism: We detect this smell in an Ansible task when the package installation lacks proper update management. If any of the following conditions are true, we detect this smell in the task [19,43,64,99].

- Missing update strategy: The task uses the package module to install a package, but it doesn't explicitly specify the update attribute with values such as upgrade or upgrade-cache. This omission may lead to the installation of outdated packages.

- No version comparison: The task doesn't perform any checks to verify if a newer version of the package is available. missing package_facts or check_mode component lead to smell.

4.2.2.3 Incompatible version dependency

This smell occurs when an Ansible playbook or role specifies a package or library version that is either no longer available in the target system's repositories, or, incompatible with other components in the configuration [31, 34, 54, 70, 98, 99, 104]. Incompatible version dependencies in Ansible scripts can lead to missing packages, cascading dependency conflicts, and version resolution problems leading to challenges in reproducible computing environments.

Example: An Ansible playbook in repository ansible-for-devops [58] has a task namely 'Install Apache, MySQL, PHP, and other dependencies'. The task installs a specific version of PHP extensions without verifying the absence of other versions in the environment. This oversight can lead to conflicts if another PHP version is already present. Specifying fixed package versions in this task can cause dependency conflicts, playbook failures due to unavailable versions, and cascading dependency issues.

Potential fix: Maintaining a compatible dependency matrix and setting the versions according to it may help avoid the smell. Often developers set latest to the state property to ensure the latest version of the package installation. While setting the state to latest might seem tempting, it can disrupt compatibility. A better approach might be to leverage tools such as pipdeptree to analyze existing dependencies and identify potential conflicts before introducing new versions. In addition, when installing any new dependencies using package managers, upgrade_cache should be used to ensure any previous version of the dependencies are removed from the environment.

Detection mechanism: We identify this code smell in Ansible tasks under the following conditions [98, 99]:

- Fixed version: The package module is used with a specific version number in the `version` attribute. Fixing versions can prevent compatibility issues but may lead to missed security updates.

- Latest Version: The package module uses `state: latest`. This ensures up-to-date packages but may introduce unexpected changes and compatibility issues.

4.2.2.3.1 Other ways of managing dependencies

Ansible dependency management offers two main approaches: pinning dependencies to specific versions and relying on the latest versions available. Both methods have their advantages and disadvantages. Pinning dependencies (specifying version numbers) ensures consistency and predictability in your Ansible environment. This is crucial for production deployments where unexpected dependency changes can break functionality. Additionally, pinning helps mitigate security vulnerabilities by locking in known-safe versions. On the other hand, using the `latest` keyword for dependencies allows developers to benefit from bug fixes and new features as they become available. This approach is suitable for development or testing environments where staying on the cutting edge is more important than strict consistency. However, it is essential to carefully monitor dependency updates to avoid introducing regressions. Also using either of these approaches do not guarantee that the script is free from reproducibility issues; hence, developers need to ensure the appropriateness of the employed dependency management approach considering their dependency review mechanism and other factors.

4.2.2.3.2 Dependency Management and Reproducibility

Ansible's strength lies in its ability to automate configurations across various systems. However, this automation hinges on the consistent availability of dependencies – modules, libraries, and plugins that tasks rely upon for proper execution. Improper dependency management is a breeding ground for reproducibility issues. Consider an Ansible script that uses an external module not explicitly declared. If the target system or development environment experiences changes to this dependency (version update, removal), the script's subsequent runs might exhibit unexpected behavior or outright failures. This inconsistency undermines the core principle of reliable automation – reproducibility.

To ensure consistent and reliable execution, explicitly define dependencies within the Ansible script.

4.2.2.4 Assumptions about environment

This smell arises when scripts rely on unverified assumptions about the environment, like the version or the type of the target machine's operating system or the availability of certain packages or libraries. General assumptions about the environment pose significant challenges in reproducing IaC scripts as they may not be accurate or applicable in all situations and environments [28, 37, 66, 85]. Ansible scripts that make assumptions about the environment, like expecting a specific OS version or pre-installed packages, can lead to unforeseen failures when encountering different environments, causing script breakage and deployment disruptions. Additionally, hidden dependencies are formed due to reliance on pre-installed software, making it dificult to understand true script requirements and hindering portability across diverse setups. This lack of adaptability and susceptibility to version drift (environments naturally update over time) ultimately undermines the core principle of reproducibility in IaC deployments.

Example: In an Ansible playbook [15] in repository chocolatey-ansible, a task with the name 'checking if the bootstrap file has been created' is using Windows-specific component without verifying this assumption about the environment.

Potential fix: Refactor scripts with parameters and conditions to adapt to different operating system versions and distributions using Ansible facts [13]. In the above example, we may use the when property to specify the required operating system as a condition to execute this task [13].

Detection mechanism: We detect this smell in the task when any of the following conditions is true [37, 51, 66].

- Operating system: Tasks using ansible_distribution or ansible_os_family variables might be distro/family specific.

- Services and configuration: Firewall, DNS, network interface, NTP, and SSH tasks modifying configurations without checking current state assume specific configurations.

- Package management: Tasks using package download modules with URL/repository keywords without any state checks.
- OS specific modules/commands: Tasks using OS-specific modules/keywords/-commands might not be portable to other systems.

4.2.2.5 Hardware specific command

This smell occurs when scripts include commands or configurations that are tightly coupled to specific hardware components, such as particular CPU architectures or GPU models. Using hardware-specific commands in scripts, tied to particular devices such as cpu, gpu, complicates environment reproduction [42,45,63,91].

Hardware-specific commands in Ansible scripts limit script portability and introduce hidden assumptions about the target hardware. This creates testing challenges and potential vendor lock-in, ultimately hindering script reproducibility across different hardware environments. Opting for generic and portable approaches ensures your scripts adapt and function reliably in diverse deployments.

Example: A task 'Install AMD GPU drivers' in repository A:Platform64 [46] configures and installs amd gpu drivers without checking the existence of this gpu in the machine.

Potential fix: An IaC script must ensure the availability of a hardware device before executing commands specific to that device. Also, checking cross-device configuration compatibility by substituting hardware-specific commands with general tasks and using Ansible variables to abstract hardware-dependent values reduce the chances of failure. For the example presented above, we may add properties such as when, debug, gather_facts to the task to handle the situations when the expected gpu is not present in the target machine using the information gathered with variables in the task [13].

Detection mechanism: Ansible tasks should ideally be hardware-agnostic to ensure broader applicability. This detection mechanism identifies tasks that rely on commands potentially specific to certain hardware, limiting their portability [42,45,91]. We detect this smell in a task under the following condition:

- Command keywords: Tasks using commands such as lspci, lshw, lsblk, fdisk, and parted, that could be hardware-dependent. The detection logic also looks for these commands within the task definition under keys such as command, shell, and raw.

4.2.2.6 Unguarded operation

An unguarded operation smell occurs when specific Ansible components that execute operating system commands are used without considering idempotency. These non-idempotent imperative commands are not automatically checked for idempotency, and using them without proper validation may result in undesirable environmental states. Performing unguarded operation without appropriate checks and error handling threatens the idempotency property of IaC scripts and leads to this smell [12, 31, 50, 60, 65, 70, 93]. These commands might cause the infrastructure state to drift over time, introduce challenges with conditional logic and testing, and break compatibility with future system versions. This ultimately leads to inconsistent deployments and unexpected behavior, undermining script reliability.

Example: A task 'Apply machine config' in repository openshift-ansible [52] uses a command to perform an unguarded operating system-level operation without properly checking the status of the system.

Potential fix: Prioritizing idempotency in task design by using error handling constructs, such as failed-when, changed-when, and rigorously validating inputs and conditions before executing critical operations may avoid this smell. For the task in the example above, we may add a changed-when statement to ensure appropriate functionality. Also, we can use an Ansible handler to gracefully handle the error if the operation does not execute correctly [13].

Detection mechanism: Unguarded operation smell threatens the idempotency property of a task. To detect this smell, we identify tasks that execute unguarded operating system commands through ansible that are not idempotent [12, 14, 50], which include using package installers without checking the existence of the package state or version, manipulating files without idempotency checks (e.g., the existence of the file or presence of a specific configuration in a file), and creating/updating users, groups, or files without idempotency considerations [17, 54, 64, 71, 92, 98].

4.2.3 Smell reference in gray literature

In our replication package we have referenced all the documents for each of the smells but in summery broken dependency chain is referenced in 16, outdated dependency in 22, incompatible version dependencyin 23, assumptions about environment in 17, hardware specific command in 12 and unguarded operation in 26 documents. In the replication package, the name, the number and the link to the documents are available.

4.2.4 Smell catalog and detection rules validation

We reached out to eleven active researchers in the field of IaC to review our ini-tial smell catalog and detection mechanisms. Participants were asked to rate each detection mechanism on a Likert scale from one to five, where five indicated the highest appropriateness for detecting the specified smell. The survey was conducted anonymously and remained open for ten days. The questionnaire can be accessed online [95].

We received six responses. The average ratings for each smell detection mechanism were as follows: 3.33 for broken dependency chain, 2.5 for outdated dependency, 3.8 for incompatible version dependency, 4.0 for assumptions about environment, 4.0 for hardware specific command, and 3.8 for unguarded operation. These results suggest that the researchers generally had reasonable confidence in our proposed catalog and detection methods.

Outdated dependency received the lowest confidence rating, with two participants indicating that it should not be classified as a reproducibility smell. In addition to the low ratings and suggestions from the participants, we realized that all dependency-related issues could be addressed by refining our definition and detection mechanism for incompatible version dependency. This made it unnecessary to introduce a sep-arate smell category, as an outdated version of a dependency can be considered an incompatible version. By detecting a fixed version specification of a dependency as a smell, we can proactively prevent that version from becoming outdated. Given the low rating and feedback, we decided to remove the outdated dependency smell from our catalog and associated tool. Consequently, outdated dependency will not be discussed further in this paper.

Summary: We conduct a comprehensive multi-vocal literature review to identify programming practices that impact the reproducibility of Ansible scripts. Using the open coding technique, we aggregate the collected information from the mlr and identify five reproducibility smells.

4.2.5 REDUSE—A Reproducibility Smell Detection tool

We developed Reduse (REproDUcibility SmEll detector)—a tool to detect reproducibility smells in Ansible scripts. This section discusses implementation aspects of the tool. Figure 4.3 shows the architecture of the tool. It consists of four key modules that we elaborate on below.

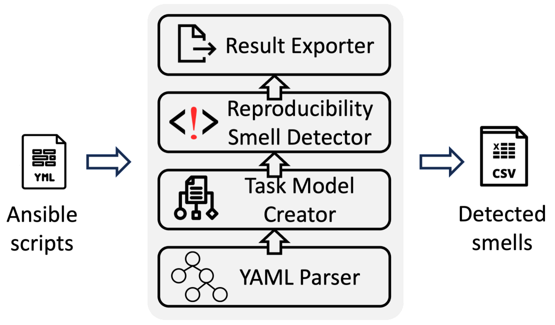


Figure 4.3: Architecture of the smell detection tool.

We can provide a path to the individual yaml file or a directory containing these files while using the tool. Reduse goes over each of the Ansible scripts individually and parses them using PyYAML[1] library. PyYAML parses the script and generates a nested list of key-value pairs. Then, the Task Model Creator module uses the parsed script and generated key-value pairs and populates our custom source code model. It is a collection of Task instances; each Task instance represents an Ansible task and contains various task properties, including task name, target hosts, and Ansible-specific properties (e.g., remote-user and gather-facts). These properties hold essential information for the tool to detect reproducibility smells. The Reproducibility Smell Detector module takes the task model instance for the Ansible script under analysis and detects reproducibility smells using the rules defined for each smell. The module also collects the associated metadata with each detected smell, such as task

[1]https://pypi.org/project/PyYAML/

name, file path, smell name, and a brief description. Finally, the Result Exporter module emits the identified smells in a csv file.

Detection process can be summarized into the following steps:

1. Parsing Playbooks: REDUSE parses Ansible YAML files using PyYAML to generate key-value pairs representing the script structure.

2. Task Model Creation: Extracted key-value pairs populate a custom model with Task instances, each representing an Ansible task with properties like name, target hosts, and Ansible-specific attributes (e.g., remote_user, gather_facts). These properties are crucial for smell detection.

3. Reproducibility Smell Detection: REDUSE employs a combination of techniques:

   (a) Rule-based matching: Predefined rules identify patterns in key-value pairs indicative of potential smells. These rules are derived from:

      - Research literature: Established research forms the foundation for smell detection rules.
      - Real-world practices: REDUSE analyzes real-world scenarios encountered by developers (good and bad practices) documented on platforms like Stack Overflow. By studying these practices, REDUSE identifies recurring patterns associated with potential reproducibility problems in Ansible scripts. This combined approach ensures a comprehensive understanding of smell characteristics.

4. Result Export: Identified smells are reported in a user-friendly format (e.g., CSV) with details like task name, file path, smell type, and a brief description, aiding developers in addressing potential issues.

4.3 Evaluation and Results

In this section, we validate our tool through manual evaluation, focusing on its effectiveness in detecting reproducibility smells in Ansible scripts. We detail our methodology, present performance metrics, and discuss the implications of our findings. Additionally, we conduct a qualitative analysis and compare our results to reproducibility challenges in other software domains to highlight the benefits of integrating our tool into development practices.

4.3.1 Tool Validation

We conduct manual validation to assess the effectiveness of the developed tool. In this section, we elaborate on the adopted method and obtained results.

Data gathering: We explore the sources of Ansible subject systems in related work and choose the oci-ansible-collection [2]. This dataset evolves 33 sub-projects responsible for many jobs such as managing network connections or creating necessary components for managing environments. In these projects, we have 84 different Ansible scripts, containing 1309 individual Ansible tasks. This dataset has been used in testing the glitch tool [86] and creating the Andromeda dataset [74].

We required repositories containing Ansible scripts to validate the tool. Additionally, we used 3 more repositories chosen from the Ansible Galaxy collections, based on the number of downloads, stars, scripts, and commits. We chose ceph_ansible, openshift_ansible, ansible_for_devops. All the projects are among the most downloaded collections on Ansible Galaxy and have more than 10 scripts, 1K stars, and 300 commits. These repositories have 174 scripts and 6722 Ansible tasks. We manually checked each repository to ensure they only contained actual Ansible scripts and not Python scripts. We explore the search options in GitHub to identify Ansible repositories; however, GitHub does not support searching repositories specific to frameworks. Moreover, given that yaml files can be used with other frameworks, such as Docker, a file-extension-based search approach also could not be used.

Methodology: Two non-author evaluators participated in the evaluation process. Both evaluators possess knowledge of IaC concepts and Ansible; they are gradu-ate students with approximately four years of software development experience. To minimize potential bias, we ensured the evaluators were only familiar with the definitions of the smells we aimed to detect and possessed a basic understanding of Ansible scripting concepts. They were not aware of the inner workings of REDUCE or the development process. Both evaluators independently assessed Ansible scripts and identified reproducibility smells. After the individual assessment was over, we matched findings from both evaluators and created a consolidated set of smells. The inter-rater agreement was high ($\kappa = 0.869$). In the case of differing opinions, they discussed and resolved such differences. We also used our tool Reduse to identify reproducibility smells in the same dataset. The results from the manual analysis and Reduse are available in our online replication package.

Table 4.1: Performance of Reduse against manually annotated ground truth

Smell	TP	FP	TN	FN	Precision	Recall	MCC	F1-score
Broken dependency chain	2,990	320	3,370	42	0.90	0.98	0.87	0.94
Incompatible version dependency	13	0	6,704	5	1.00	0.72	0.77	0.85
Assumptions about environment	2,092	780	3,710	140	0.73	0.94	0.65	0.82
Hardware specific command	10	0	6,712	0	1.00	1.00	1.00	1.00
Unguarded operation	2,780	298	2,340	10	0.93	0.96	0.89	0.94
Total	7,885	1,398	9,426	197	0.849	0.976	0.837	0.908

Evaluation: We compared the tool-generated results with manually curated ground truth. Table 4.1 presents the results of the evaluation using typical metrics: precision, recall, F1-score, and Matthews Correlation Coeficient (mcc). The tool performs well with F1-score = 0.908 and mcc = 0.837.

Our tool validation revealed a relatively high rate of false positives for broken dependency chain and assumptions about environment. We manually checked a sample of the reported false positive instances and updated the identification rule for broken dependency chain, reducing these errors. The tool incorrectly identifies broken dependency chain because it checks whether a task ensures its correct execution via specific Ansible attributes (e.g., set-facts, package-facts, and assert) within the task. Updating the corresponding rules reduced the number of false positives. Also, since the tool currently does not support analysis across tasks, it reports false positive instances. Additionally, typically static analysis tools, such as Reduse, rely on detection rules that are generic in nature. Given that there are numerous ways to implement any desired behavior and the unavailability of all the contextual information, such as business or technical constraints, the rules result in false positives. Similarly, the tool lacked additional detection rules (e.g., usage of firewall, usage of ssh) for catching assumptions about environment smells. The tool reports perfect F1-score and mcc for the rest of the smells. Finally, smells are indicators of deeper issues and not definitive errors; developers can consider identifying a potential issue to decide whether a reported issue requires further exploration.

4.3.2 Empirical study

This section focuses on deriving results to address our research questions. We will first delve into data processing techniques before detailing our approach for each research question, ensuring a data-driven investigation.

4.3.2.1 Data collection

As we discuss in Section 4.3.1, GitHub does not offer a convenient means to select a subset of Ansible repositories. We rely on Ansible Galaxy [1]—a hub for hosting, searching, and sharing Ansible projects for identifying our subject systems. Ansible Galaxy divides the hosted repositories into nine categories (i.e., System, Networking, Database, Packaging, Security, Development, Cloud, Monitoring, and Web). We apply selection criteria to identify the repositories for our empirical analysis. First, we select 100 most downloaded repositories from each category. The Ansible Galaxy platform does not provide a developer-friendly mechanism (e.g., platform apis similar to GitHub apis) to extract the required information (in our case, a GitHub repository link as part of the metadata of the search Galaxy's result).

To overcome the challenge, we developed a Python script to extract GitHub repository links from the web search results on Ansible Galaxy. We also use GitHub repository metadata to filter out low-quality and unmaintained repositories among the initial 900 repositories. Specifically, we select repositories with a minimum of 50 commits, at least five stars, and the last commit no older than a year. With these criteria in place, our final dataset comprises 290 repositories; these repositories contain $4,100$ Ansible scripts and $19,412$ distinct Ansible tasks. After cloning each repository, we manually checked the files, ensuring that we only have Ansible scripts in the project, not the files that contain Python scripts. We also discard the testing Ansible scripts. One important reason is that hard-coded values are commonly used for testing purposes but can be detected as a smell by our tool.

4.3.2.2 Results of RQ2

RQ2 aims to understand the proliferation of reproducibility smells in open-source repositories.

4.3.2.2.1 Approach To answer this research question, we use Reduse to identify the presence of reproducibility smells in all selected repositories for each task of their playbooks. The tool identifies smells and stores the identified instances with relevant metadata (i.e., repository name, script path, task name, the identified smells, and a brief description) for each analyzed Ansible script. We aggregate all the smell instances across all the analyzed repositories to calculate smell frequency across all tasks.

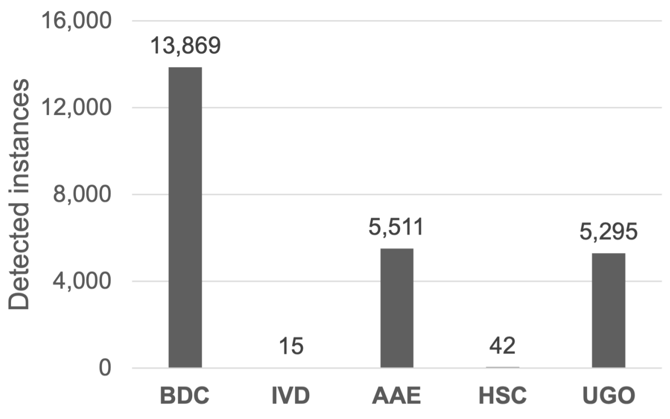


Figure 4.4: Frequency of detected reproducibility smells in Ansible tasks. BDC refers to broken dependency chain, IVD to incompatible version dependency, AAE to assumptions about environment, HSC to hardware specific command, and UGO refers to unguarded operation.

4.3.2.2.2 Results Figure 4.4 shows the total number of reproducibility smell instances detected in all the analyzed Ansible tasks. broken dependency chain is the most frequently occurring smell. A high frequency of this smell indicates that software developers do not ensure the existence of all the required dependent packages in their infrastructure specifications. Similarly, assumptions about environment and unguarded operation smells show a high frequency, indicating that infrastructure specifications are written without concern for the portability of the instructions across various execution environments and without ensuring the idempotency properties of the specified operations. Onthe other hand, incompatible version dependency, and hardware specific command smells have the lowest frequency. This suggests that software developers rarely specify outdated dependencies, or specify operations that require hardware-specific configurations.

4.3.2.3 Results of RQ3

This research question investigates the relationships among reproducibility smells, specifically pair-wise correlation and co-occurrence. Analyzing how reproducibility smells co-occur in Ansible scripts goes beyond identifying individual issues. It reveals underlying patterns and root causes that lead to multiple smells. This knowledge helps developers make targeted improvements. Instead of fixing each smell separately, they can address the root cause, leading to more eficient remediation.

4.3.2.3.1 Approach The process starts with a list of identified smells using Re-duse over all Ansible scripts. Then, we consolidate these smell instances at the repository granularity, i.e., each row represents the total number of smells per smell type for a repository. We use Spearman correlation analysis on each pair of smells to find the correlation between all pairs of reproducibility smells. To determine whether two reproducibility smells occur together in a task, we carry out a fine-grained analysis at the Ansible task granularity.

We create a contingency matrix [100] for each pair of smells, as shown in Table 4.2, and compute the φ coeficient [105], as specified below. The φ coeficient provides a measure of co-occurrence, sensitive to both the presence and absence of reproducibility smells within individual tasks.

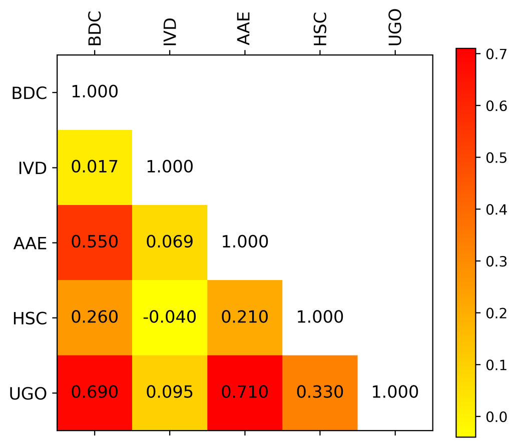
		Smell B	
		Present	Not present
Smell A	Present	a	b
	Not present	c	d

Table 4.2: Contingency matrix for a smell pair.

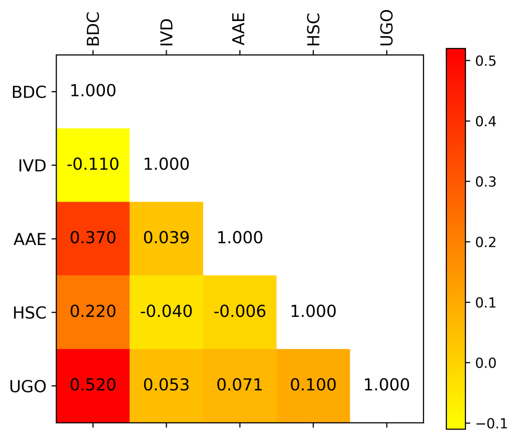
$$\varphi = \frac{a \times d - b \times c}{\sqrt{(a + b) \times (c + d) \times (a + c) \times (b + d)}} \tag{4.1}$$

Results: First, we use the Shapiro-Wilk test [88] to check the normality of the data distribution. We perform the test on each of the detected smells in all of the scripts; we obtain w in the range of 0.01–0.61 with the p-value < 0.05 for all the observations. The results of the test inform us that the data is not following a normal distribution. Therefore, we use Spearman correlation to measure the correlation between two reproducibility smells.

Figure 4.5a presents the correlation coeficients between pairs of reproducibility smells; all the observations are statistically significant with p-values < 0.05.



(a) Spearman correlation coeficients with absolute smell count.



(b) Spearman correlation coeficients with normalized smell count.

Figure 4.5: Correlation analysis.

We observe a high positive correlation between unguarded operation and assumptions about environment smells. The unguarded operation also exhibits moderate to high correlation with broken dependency chain. and assumptions about environment has a moderate correlation with outdated dependency and broken dependency chain smells. These high to moderate correlations among unguarded operation, assumptions about environment, and broken dependency chain suggest that if a repository has a large number of one kind of smell, it is likely to find other kinds of smells among the other smells with a high to moderate correlation.

The number of detected smell instances plays a role in the correlation analysis. Specifically, smells unguarded operation, assumptions about environment, and broken dependency chain show a high correlation with other types of smells as they are frequently detected. On the other hand, hardware specific command and incompatible version dependency smells that are the least frequently detected ones naturally show a low correlation with other smells. The size of the repository, i.e., the number of tasks in a repository, may confound the analysis. To remove the factor of size from the analysis, we compute the normalized number of smells by dividing the total number of smells by the total number of Ansible tasks in a repository. We obtain a new set of correlation coeficients for the normalized smell count that we show in Figure 4.5b. The analysis shows an interesting observation. The erstwhile high correlation between, for example, unguarded operation and assumptions about environment smells is no longer visible. It implies that the high correlation was only due to the size of the repositories. With the normalized smell count, the smell pair unguarded operation and broken dependency chain shows the highest correlation.

We investigate the co-occurrence relationship between pairs of reproducibility smells at the fine-grained task granularity. The co-occurrence relationship shows whether two smells occur together at the task granularity, whereas correlation captures the tendency and proportion of smells to be detected for all the tasks in a repository. Figure 4.6 shows the calculated φ coeficient for each smell pair. The co-occurrence relationship is directional, unlike correlation, i.e., co-occurrence between (a,b) is not equivalent to (b,a).

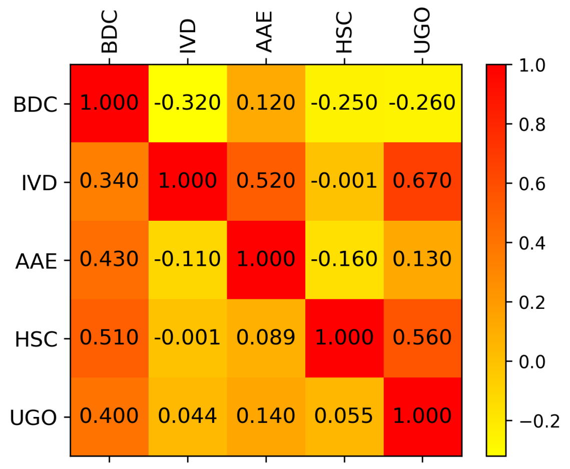


Figure 4.6: Smell co-occurrence at the Ansible task granularity.

We observe that the unguarded operation smell shows a high co-occurrence with incompatible version dependency. These smells usually occur when using package installers in an inappropriate way of specifying the package's version. Similarly, assumptions about environment smell exhibits a moderate degree of co-occurrence with incompatible version dependency. One potential reason for this relationship is that a script that assumes an execution environment may specify hard-coded versions for the required packages. The hardware specific command smell does not co-occur with other smells.

Analyzing how reproducibility smells tend to co-occur within Ansible scripts offers valuable insights that extend beyond simply identifying individual smells. By examining co-occurrence, we can uncover underlying patterns and potential root causes that lead to multiple smells appearing together in a script. This knowledge empowers developers to make more targeted improvements. Instead of fixing each smell independently, they can focus on addressing the root cause that triggers multiple smells simultaneously, leading to more eficient remediation efforts.

Furthermore, understanding co-occurrence patterns is valuable for vendors creating improved smell detection tools. By tailoring detection algorithms to identify frequently co-occurring smells, tools can become more effective in pinpointing potential problems within Ansible scripts.

Finally, researchers developing frameworks to guide Ansible script development can leverage co-occurrence knowledge to inform the design process. Frameworks can be refined to address vulnerabilities that lead to the co-occurrence of specific smells, promoting more robust script construction from the beginning.

Summary: Our correlation analysis uncovers significant positive correlations between specific reproducibility smells, implying that repositories with one such smell tend to exhibit others. The deeper analysis with the normalized smell counts shows that broken dependency chain and unguarded operation smells are moderately correlated. The co-occurrence analysis reveals the high to moderate co-occurrence tendency between specific smell pairs. Co-occurrence analysis findings offer valuable insights for both researchers and tool vendors. Researchers can use this information to refine framework design, while tool vendors can improve their tools' ability to detect reproducibility issues more effectively.

4.3.2.4 Key takeaway from RQ2 and RQ3

The high frequency of reproducibility smells like broken dependency chain, assumptions about environment, and unguarded operation indicates common pitfalls developers face in ensuring their infrastructure specifications are both portable and dependable across different environments. These findings highlight the need for better tools and practices to address these reproducibility challenges in Ansible scripts.

4.3.2.5 Qualitative Analysis

In this section, we elaborate on the results obtained, extend our analysis through qualitative methods, and discuss the implications of our findings. We conducted a qualitative analysis focusing on the manifestation of reproducibility smells in real-world Ansible projects. We selected the top 20 most downloaded Ansible repositories from Ansible Galaxy, filtering for those with at least 100 commits and ten reported issues on GitHub. Special attention was given to repositories used in production environments by notable companies like Cisco or RedHat. Each selected project was manually reviewed to ensure issues were attributable to Ansible scripts rather than underlying Python code.

We manually reviewed a total of 152 issues across eight selected repositories. From these, 28 issues were directly linked to reproducibility aspects discussed in this paper. This significant proportion underscores the critical impact of reproducibility smells on operational stability and the importance of early detection and remediation. Detailed findings and mappings of reproducibility smells for these issues are documented in our replication package [9].

For example, in the cisco.nxos repository [25], used for managing NX-OS network appliances, issues #801 and #803 arose from not using appropriate Ansible components when making changes in the files and not properly checking the state of the file after each change that led to idempotency violations which is detected as unguarded operation smell. Similarly, issue #542 in the same repository demonstrated failure due to unguarded use of the command attribute without adequate status change measures, a classic instance of Idempotency. These examples underscore the practical application of our tool (Reduse) in identifying and mitigating reproducibility issues early in the development lifecycle.

Cisco asa repository reports an issue [26] where a new version of a module was not compatible with another dependent module resulting in unexpected output in execution. It indicates incompatible version dependency smell that is caused by specifying a fixed version for a module without checking their compatibility. Another issue from the same repository [27] highlights a problem with an Ansible task that failed to verify its proper execution using appropriate Ansible components such as assert. This oversight led to errors in playbook execution, resulting in incorrect configuration of the environment based on the configuration file. Such issues can be captured using the broken dependency chain smell.

4.3.2.6 Implications

Our study not only quantifies the prevalence of reproducibility smells in active Ansible repositories but also emphasizes their implications in terms of operational reliability and maintenance overhead. By leveraging tools like Reduse, developers can proactively identify and refactor reproducibility smells, thereby enhancing the robustness and maintainability of Ansible playbooks across diverse deployment environments. The qualitative analysis reinforces the need for rigorous adherence to best practices in Ansible playbook development. It advocates for comprehensive dependency management, thorough validation mechanisms, and continuous refinement of scripting practices to minimize the occurrence of reproducibility issues. Moreover, it underscores the role of automated tools in promoting consistent and reliable infrastructure management practices.

Reduse offers substantial benefits in improving the quality and reliability of Ansible playbook development. By integrating Reduse into CI/CD pipelines, developers can conduct early static code analysis to preemptively detect reproducibility issues. This integration prevents issues from propagating to production environments and fosters a culture of continuous improvement in playbook development practices.

Additionally, Reduse serves as an educational tool for new developers, guiding them towards writing robust and maintainable Ansible playbooks. Its capability to identify and categorize reproducibility smells provides actionable insights for developers to enhance their scripting practices and contribute to the overall reliability of infrastructure deployments.

Chapter 5

Sustainable AI Inference on Edge Devices

5.1 Methodology and Deployment

EdgeAI is increasingly utilized in applications that demand real-time processing, low latency, and eficient resource management. Typical applications include smart buildings, IoT systems, image and video processing in autonomous vehicles, and natural language processing in voice assistants. Each of these applications relies on specific models tailored to their unique needs. However, edge devices have limited resources (computing, memory, and power) compared to their cloud counterparts. Thus, assessing the deployment feasibility of AI models on resource-constrained edge devices is essential to shed light on their performance and resource usage while hosting a diverse range of models. The primary aim of this research is to evaluate the feasibility of deploying AI models on edge devices, identify the associated challenges, and to discover techniques for utilizing AI on resource-constrained devices.

The assessment workflow depicted in Figure 5.1 starts with selecting appropriate AI models tailored for edge devices, categorized based on their application domains. In parallel, we choose a set of edge devices (e.g., NVIDIA Jetson Nano) and their respective AI frameworks (e.g., TensorFlow Lite). The next step is to train the chosen models in the legacy cloud servers using their optimal hyperparameters for the chosen platform. The trained models must be converted to their lighter version based on the chosen device and platform. We deploy the converted models on the chosen devices for performance evaluation, which includes measuring models' accuracy, inference time, power and memory consumption. In the following, we elaborate on each step of the proposed assessment methodology.

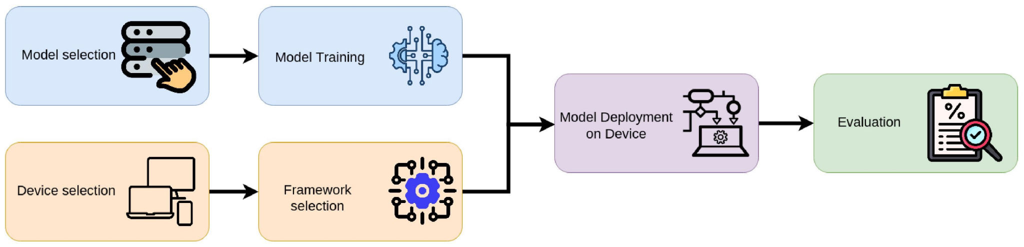


Figure 5.1: The workflow of AI inference assessment in edge devices.

5.1.1 Model Selection

Given that most common AI applications on edge devices fall under the image processing or image classification category, we employ a set of traditional ML and neural network models. We added large language models as an additional category to cover possible usage in voice assistants. We present a list of models from each category below.

- Traditional ML Models: Decision Trees, SVM, and KNN are well-suited for quick decision-making in edge environments like smart buildings and IoT systems, where real- time sensor data analysis is crucial [103].
- Neural Network Models: ANN, CNN, R-CNN, ResNet-50, and MobileSSD are widely used for processing image and video data. These models are commonly used in edge applications such as gesture recognition, dynamic environment adaptation, autonomous driving, and video analytics, where high accuracy and real-time performance are essential [103].
- Large Language Models: We choose TinyBERT and Phi-2-orange optimized for natural language processing on edge devices. These models are widely used

in voice assistants, text processing, and other NLP-driven applications, showcasing the versatility of edge computing [84].

We ensure compatibility with the selected devices and frameworks by aligning each model category with specific edge computing applications. This approach facilitates a comprehensive exploration of performance metrics, including accuracy, inference speed, memory usage, and energy consumption across diverse edge scenarios.

5.1.2 Device Selection

We review existing literature in edge computing for selecting representative hardware devices and select the following 3 devices with unique characteristics and advantages. Note that there are other edge devices, e.g., Google Coral USB, that can offer good performance on the edge; our selection of these devices is based on their widespread adoption in research and practical applications. The Raspberry Pi's affordability and flexibility, coupled with the Jetson Nano's GPU capabilities and the Neural Compute Stick's specialized inference acceleration, offer a diverse set of platforms for our edge computing experiments.

- Raspberry Pi: The Raspberry Pi is widely recognized for its versatility and cost-effectiveness in edge computing applications [84]. Its compact size, low power consumption, and General Purpose Input/Output (GPIO) enable seamless interfacing with a wide range of sensors and peripherals, making it ideal for Internet of Things (IoT), robotics, and educational projects [7].

- Intel Neural Compute Stick: The Intel Neural Compute Stick is a compact USB device designed to accelerate deep neural network inference at the edge. Equipped with the Intel Movidius Myriad X Vision Processing Unit, it offers high-performance inference capabilities while consuming minimal power. This makes it suitable for real-time applications like image recognition and speech processing in edge environments [3].

- Nvidia Jetson Nano: The Nvidia Jetson Nano Developer Kit features a CUDA-capable GPU and ARM Cortex-A57 CPU, providing robust computing power for deploying AI applications at the edge [103]. It supports popular

AI frameworks such as TensorFlow and PyTorch, making it suitable for tasks ranging from autonomous robots to intelligent surveillance systems [4].

5.1.3 Framework Selection

We first choose TensorFlow Lite, PyTorch Mobile and MXNet as the candidate edge platforms due their wide adoption in academia and industry. MXNet is a deep learning framework known for its scalability and eficiency, particularly in distributed training and inference across multiple GPUs. However, MXNet's primary strengths lie in its capability for handling large-scale training tasks and specific use cases requiring extensive parallelism, which may not be as critical in edge deployments. PyTorch Mobile integrates seamlessly with PyTorch, offering dynamic computation graphs and flexibility for edge applications. However, we select TensorFlow Lite as the main platform in our assessment due to the following reasons.

- Optimized Inference and Training: TensorFlow Lite ensures eficient inference with advanced optimization techniques like quantization and GPU support, while TensorFlow provides optimized implementations for initial model training [23, 84].
- Broad Platform Support: TensorFlow Lite supports a wide range of platforms, including ARM-based devices, Android, iOS, and micro-controllers, making it versatile for deployment across diverse hardware.
- Stable and Proven Deployment: TensorFlow and TensorFlow Lite offer reliable and stable deployment solutions, consistently used in research and industry, ensuring trust and flexibility.
- Comprehensive Deployment Tools: TensorFlow Lite simplifies model conversion and deployment with tools like TensorFlow Lite Converter, facilitating smooth transitions from training to deployment [78, 107].

Variants of TensorFlow Lite: The lighter frameworks are different on different edge devices with respective target-specific optimizations. For example, Nvidia Jetson uses TensorRT, an SDK developed by Nvidia that optimizes deep learning models for inference on Nvidia GPUs. TensorRT applies several advanced techniques such as precision calibration (e.g., FP16 and INT8 quantization), layer fusion, and kernel auto-tuning to accelerate model inference, significantly improving both performance and eficiency while reducing latency.

Similarly, the Intel Neural Compute Stick utilizes Intermediate Representation (IR) format of the models, which are a crucial component of the Intel OpenVINO toolkit. The IR format consists of an XML file that describes the network structure and a binary file containing the model weights. These IR models are optimized for Intel hardware by applying transformations such as weight pruning, quantization, and operator fusion, allowing for eficient inference on devices like the Neural Compute Stick. By tailoring frameworks like TensorRT for Nvidia Jetson and IR format of the models for Intel devices, we maximize performance by leveraging specific hardware optimizations, ensuring that our AI models run eficiently and effectively on the target hardware.

5.1.4 Model Training

The training process varies depending on the category of models, which we elaborate below.

KNN, SVM, DT, linear classifiers, ANN, CNN, FFNN and R-CNN: We train these models using standard datasets like MNIST [67] over the legacy framework like TensorFlow. The training process involves feature extraction, data normalization, and parameter tuning to optimize model parameters for the best accuracy. We use the trained for multi-class classifications.

Resnet-50 and MobileSSD: In this case, the training process involves data augmentation, regularization techniques such as dropout, and optimization strategies like learning rate scheduling. These models are trained on respective standard dataset ImageNet [53] using TensorFlow. We use the trained for multi-class classifications.

TinyBERT and phi-2-orange: In large language models, TinyBERT is trained using the GLUE [102] dataset, specifically tailored for question answering tasks. The training process involves fine-tuning a pre-trained BERT model with additional layers suited for GLUE tasks. This includes adjusting hyper-parameters like learning rate and batch size to enhance the model's ability to handle question answering effectively.

The phi-2-orange model is trained on the OpenAssistant dataset Oasst1 [62], which is used for intent classification in conversations. The training involves processing labeled prompts indicating various intents (e.g., informing, questioning, complaining). The model is fine-tuned to categorize user prompts accurately by leveraging transfer learning techniques and optimizing classification performance on the dataset.

The training process of the above models is summarized in Table 5.1.

Model	Training Framework	Target Device	Dataset	Training Information
KNN, SVM, DT, Linear Classifier	TensorFlow	★, ✳, ☆	MNIST	Lightweight models optimized for low-power edge devices.
ANN, CNN, FFNN, R-CNN	TensorFlow Keras	★, ✳, ☆	MNIST	Quantization and pruning applied for enhanced inference speed.
ResNet-50, MobileSSD	TensorFlow Keras	★, ✳, ☆	ImageNet	Mixed precision (FP16) for improved performance.
TinyBERT	TensorFlow Keras	★, ✳, ☆	GLUE	Model distillation for eficient edge inference.
Phi-2-Orange	TensorFlow Keras	★, ✳, ☆	OpenAssistant	Optimized for performance on powerful edge devices.

Table 5.1: Models, Training Tools, Target Devices, Datasets, and Training Information for Edge Devices. Symbols used for target devices: ★ - Raspberry Pi, ✳ - Intel Stick, ☆ - Jetson Nano.

Table!5.2, 5.3, 5.4, 5.5 present hyper-parameters for training the models.

Model	Hyperparameters
KNN	- Neighbors: 3 - Data Precision: 16-bit
DT	- Data Precision: 16-bit - Random State: Fixed
SVM	- Gamma: 0.001 - Data Precision: 16-bit
Linear	- Optimizer: SGD - Loss Function: Categorical Cross-Entropy - Epochs: 5 - Batch Size: 64 - Validation Split: 20% - Quantization: FP16 for TensorFlow Lite

Table 5.2: Hyperparameters of KNN, Decision Tree (DT), SVM, and Linear Classifier.

We save all trained models H5 file format to preserve both their weights and architecture. This step is essential for eficiently converting the models into TensorFlow Lite format, making them suitable for deployment on edge devices [23, 84, 107]. Following this structured approach ensured consistency and compatibility across our model implementations, establishing a solid foundation for seamless deployment and integration into edge computing environments.

Model	Hyperparameters
FFNN	- Architecture: 2 Dense Layers (128, 64 units) with ReLU - Output Layer: 10 units with Softmax - Optimizer: Adam - Loss Function: Categorical Cross-Entropy - Epochs: 5 - Batch Size: 64 - Validation Split: 20% - Quantization: FP16 for TensorFlow Lite
CNN	- Layers: 2 Conv Layers (32, 64 filters) with 3x3 kernels - Pooling: 2 Max-Pooling layers (2x2) - Dense Layer: 128 units with ReLU - Output Layer: 10 units with Softmax - Optimizer: Adam - Loss Function: Categorical Cross-Entropy - Epochs: 5 - Batch Size: 64 - Quantization: FP16 for TensorFlow Lite
R-CNN	- Conv Layers: 3 Layers (32, 64, 64 filters) with 3x3 kernels - Pooling: 3 Max-Pooling layers (2x2) - Dense Layer: 256 units with ReLU - Output Layer: 10 units with Softmax - Optimizer: Adam - Loss Function: Categorical Cross-Entropy - Epochs: 5 - Batch Size: 32 - Quantization: FP16 for TensorFlow Lite
ANN	- Architecture: Custom Layer Configuration - Optimizer: Adam - Loss Function: Categorical Cross-Entropy - Epochs: 10 - Batch Size: 64 - Validation Split: 20%

Table 5.3: Hyperparameters of FFNN, CNN, R-CNN, and ANN.

5.1.5 Evaluation Metrics

We collect the following metrics while experimenting with the selected models.

5.1.5.1 Memory Utilization

Memory utilization is a critical metric for evaluating eficiency in resource-constrained environments, as it reflects the total memory resources consumed during inference. It covers both device memory (e.g., GPU memory on the Nvidia Jetson Nano) and

Model	Hyperparameters
Resnet-50	- Architecture: 50 layers with GlobalAveragePooling2D - Dense Layers: 1024 units with ReLU - Image Size: 224x224 pixels - Optimizer: Adam - Loss Function: Sparse Categorical Cross-Entropy - Batch Size: 32 - Transfer Learning: Pre-trained ImageNet weights with frozen layers
MobileSSD	- Architecture: Depthwise Separable Convolutions - Image Size: 224x224 pixels - Optimizer: Adam - Loss Function: Sparse Categorical Cross-Entropy - Batch Size: 32 - Transfer Learning: Pre-trained ImageNet weights with frozen layers

Table 5.4: Hyperparameters of ResNet50 and MobileSSD.

host memory (system RAM used by the CPU). Device memory is optimized for fast data processing by specialized hardware, while host memory handles general system tasks and data transfers. By focusing on total memory usage from the host system's perspective, our analysis ensures consistent measurement across platforms like the Nvidia Jetson Nano, which uses unified memory, and the Raspberry Pi, which relies solely on CPU memory.

5.1.5.2 Accuracy

Accuracy measures the proportion of correctly classified instances out of the total evaluated instances, indicating the model's performance in predicting correct class labels. Accuracy is computed by dividing the number of correctly classified instances by the total number of instances in the dataset and expressing it as a percentage:

$$\text{Accuracy} = \frac{\text{Number of Correctly Classified Instances}}{\text{Total Number of Instances}} \times 100\%$$

A high accuracy indicates accurate model predictions, while a low accuracy suggests the need for model improvements.

5.1.5.3 Inference Time

Inference time specifically measures the time taken by the model to process inference. Inference time is crucial for real-time or latency-sensitive applications. Total

Model	Hyperparameters
TinyBERT	- Learning Rate: 2×10^{-5} - Epochs: 3 - Optimizer: Adam - Loss Function: Sparse Categorical Cross-Entropy - Dataset: GLUE - Tokenizer: BertTokenizer - Deployment: TensorFlow Lite
Phi-2-orange	- Batch Size (Training): 16 - Batch Size (Evaluation): 64 - Epochs: 3 - Warm-up Steps: Configured - Weight Decay: Configured - Dataset: OpenAssistant - Managed by: Hugging Face Trainer

Table 5.5: Hyperparameters of TinyBERT and Phi-2.

execution time refers to the overall time taken by the pre-processing, inference, and post-processing steps. Both inference and total time are typically measured by logging timestamps before and after each operation and calculating the differences.

5.1.5.4 Power Consumption

Power consumption during the inference phase is a vital metric for assessing energy eficiency, particularly in battery-powered or energy-constrained environments. To measure power consumption accurately, hardware instruments such as power meter or software tools interfacing with hardware sensors are used. These tools can capture real-time power data by connecting to the device or utilizing built-in sensors that monitor voltage and current. For accurate measurements, it is essential to isolate the inference phase from other activities and background processes, ensuring that the recorded power consumption reflects only the energy expended during inference. Inference energy consumption is measured by obtaining a total energy consumption report for each input sample during the inference, with logged timestamps to accurately capture the duration.

5.2 Implementation and Deployment

Figure 5.2 depicts the implementation and the deployment process of our measurements. We elaborate each step below.

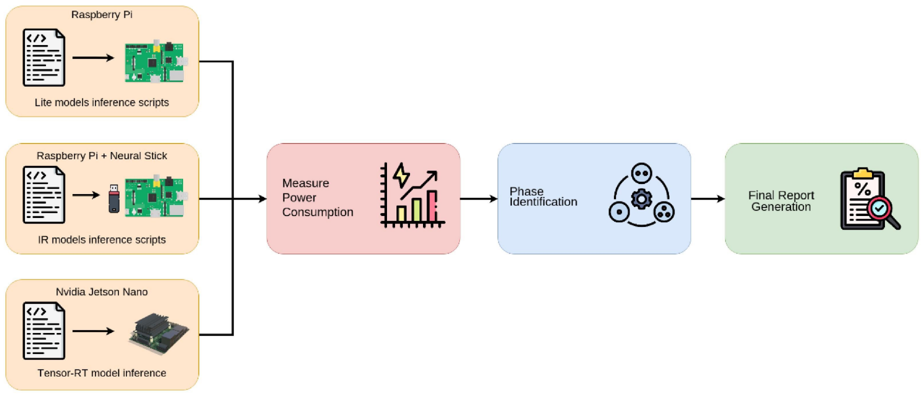

Raspberry Pi
Lite models inference scripts
Raspberry Pi + Neural Stick
IR models inference scripts
Nvidia Jetson Nano
Tensor-RT model inference
Measure Power Consumption
Phase Identification
Final Report Generation

Figure 5.2: Deployment and measurement setup.

5.2.1 Device Setup Process

We first prepare each platform for deploying the chosen models, performing inference, and conducting intended measurements.

5.2.1.1 System Software Deployment

Setting up a Raspberry Pi for running TensorFlow Lite model inference involves flashing the latest Raspberry Pi OS (64bit), updating and upgrading packages such as python-pip3, tflite-runtime, psutil and creating a virtual environment. TensorFlow Lite and its dependencies are installed within this virtual environment. In the case of Intel Neural Stick with a Raspberry Pi, we install Intel Neural Stick dependencies in addition to the standard Raspberry Pi setup. After creating a virtual environment and installing TensorFlow Lite, we used OpenVINO alongside TensorFlow Lite inference scripts for eficient model deployment and execution on the Raspberry Pi. Setting up the Nvidia Jetson Nano involves installing the necessary requirements for executing TensorFlow Lite inference. The Jetson Nano is powered up and connected to the Internet, followed by updating system packages and upgrading packages such as python-pip3, tflite-runtime, psutil. We also install any other required Python libraries.

5.2.1.2 Framework Deployment

TensorFlow Lite framework comes with a Converter tool to convert a saved model in H5 format to the TensorFlow Lite format, which is saved as a (.tflite). This file can be deployed on Raspberry Pi and Nvidia Jetson Nano. We use OpenVINO to convert the models in TensorFlow Lite format into Intermediate Representation (IR) format to be deployed in the Intel Neural Compute Stick. OpenVINO provides a comprehensive set of tools and libraries for eficient deployment and inference across various Intel architectures, ensuring our machine learning models can fully leverage Intel hardware acceleration. Finally, we use TensorFlow library along with TF-TRT, which is an integration within TensorFlow that allows for the optimization of TensorFlow models using NVIDIA's TensorRT, to convert the saved TensorFlow models to the models in TensorRT format.

5.2.1.3 Model and Data Deployment

Once the edge devices are ready with the desired platforms, we deploy a measurement script to perform the inference of the chosen models. The script consists of five steps: initialization, loading test dataset loading model image, performing inference, and collecting performance metrics.

The TensorFlow Lite interpreter in Raspberry pi loads the model image and the test dataset into the memory. This process typically involves iterating through the dataset, feeding each sample to the model, and capturing output predictions for further analysis.

For the Nvidia Jetson Nano,in addition to using the TensorFlow Lite models, we converted the Lite models to Tensor-RT models optimized for the device. After conversion, the model is loaded onto the device, and inference is performed. When using a model converted with OpenVINO for inference, the Inference Engine component is utilized to optimize and execute the model on Intel hardware. For the Intel Neu-ral Compute Stick, the Inference Engine exploits hardware acceleration for enhanced performance. The model is loaded onto the Neural Compute Stick, and inference is performed directly on the device, leveraging its parallel processing capabilities.

5.2.2 Performance Measurement and Report Generation

We measure various performance metrics: inference accuracy, inference time, memory usage, and power usage. We use a batch size of 25 images to measure the performance of the chosen models. We repeat each measurement 10 times except for resnet-50 and mobileSSD, which is the average of 5 repetitions. Finally, we executed the LLM models once. In the following, we present the measurement process using appropriate tools.

Inference Accuracy and Time Measurement: We measure the inference accuracy using the function that compares the predictions with the true labels as TensorFlow Lite, OpenVINO, or TensorRT cannot provide a single function to measure inference accuracy directly. The obtained accuracy is then saves in a log file. Start and end timestamps for each stage are logged to calculate total and inference times. The Python time module is used to measure and log time because it is simple, precise, and part of Python's standard library. The code snippet logs the start and end time of a phase using time.strftime() to format the current time. For measuring elapsed time, time.time() is used to record the start and end times, and the difference between them gives the total duration.

Memory Usage Measurement: We use psutil library to measure memory utilization, focusing on the total RAM consumption during script execution. psutil provides detailed insights into the memory usage of both individual processes and the overall system, allowing us to monitor the impact of our scripts on system RAM.

Energy Measurement: Firstly, establishing a consistent method for measuring energy consumption proved a significant hurdle. This is because the software-based energy measurement packages are unavailable across the chosen hardware devices. We need a robust tool for all three devices. However, as shown in Table 5.6, no software package can be used for all the devices; thus, we choose a hardware-based tool, namely USB power meter.

Device	Tool Type	Tool Name	Summary
Raspberry Pi	Software	PiJuice API	Monitors power consumption with PiJuice HAT.
	Software	PowerAPI	Monitors power consumption on Pi and peripherals.
	Hardware	USB Power Meter	Measures real-time voltage, current, and power.
	Hardware	INA219/INA226 Power Monitor	I2C module for monitoring power on rails.
Intel Neural Compute Stick	Software	Intel® Power Gadget	Provides power insights for Intel hardware.
	Software	psutil	Monitors system resources for indirect power analysis.
	Hardware	USB Power Meter	Measures power usage via USB.
Nvidia Jetson Nano	Software	tegrastats	Native tool for real-time power monitoring.
	Software	Jetson Power Monitor	Logs power data from INA3221 chip.
	Hardware	USB Power Meter	Measures power usage via USB.

Table 5.6: Energy consumption measurement tools for Raspberry Pi, Intel Neural Compute Stick, and Nvidia Jetson Nano.

The USB meter's output is connected to the device under test, such as a Raspberry Pi, ensuring the input matched the device's power supply. After recording power consumption using the USB power meter, we use its PC application to obtain measurement reports. We record the power consumption for the entire script execution time then using our logged timestamps, we identify timestamps for each step and calculate power consumption for the inference.

Before each script execution, unnecessary processes are terminated, and the virtual environment is reactivated to prevent caching effects. The power meter continuously records power consumption at a sampling rate of 16 samples per second. After the script execution, power consumption records and corresponding timestamps are exported. We include an initialization phase when running scripts to measure the device's power consumption at near-idle states. We average these values and subtract them from the power consumed during the inference phase. This approach isolates the power used specifically by the inference operations, excluding the device's idle power and other components.

5.3 Evaluation Results

This section presents the evaluation results answering the three research questions.

5.3.1 How do learning models perform on selected edge devices?

5.3.1.1 Traditional ML Models

Table 5.7 presents the observed results for all considered traditional ML models. In the table, we present the mean value of the measured metric along with 95% confidence interval. As shown in table, the evaluation of K-Nearest Neighbors (KNN), Support Vector Machine (SVM), Decision Tree (DT), and Linear Regression models across various hardware platforms reveals how different devices handle these models. Nvidia Jetson Nano consistently outperforms the Raspberry Pi in inference speed and execution time due to its powerful hardware capabilities, such as GPU, which accelerates parallel computations. This makes Jetson Nano effective, particularly for models like SVM and KNN, which require intensive mathematical operations and benefit from hardware acceleration. The SVM model on Jetson Nano is the top performer, leveraging its hardware capabilities to handle the model's linear operations far more eficiently than the Raspberry Pi, which lacks a dedicated GPU. For KNN, Jetson Nano's parallel processing capabilities allow for faster distance calculations, though it demands more memory and energy. The Decision Tree model consistently performs across devices since it involves simpler, sequential decisions that do not benefit much from hardware acceleration, allowing the Raspberry Pi to perform comparatively well. Linear Regression, being less computationally intensive, also shows a smaller performance gap between devices, with Jetson Nano having a slightly better performance.

5.3.1.2 Neural Network Models

Table 5.8 presents the observed metrics along with their confidence interval for all considered devices and models. As shown in table, the performance of neural network models across different hardware platforms highlights their strengths and suitability for specific tasks. Nvidia Jetson Nano, particularly when optimized with TensorRT, consistently outperforms Raspberry Pi and Intel Neural Stick in terms of inference speed and accuracy. For example, when running a CNN, the Jetson Nano with TensorRT offers the best inference and total times, making it the most eficient platform for real-time applications. On the other hand, Raspberry Pi, while slightly slower,

Table 5.7: Performance of traditional ML models. Metrics are reported along with their confidence interval in square brackets.

Model	Mean Accuracy (%)	Inference Time (s)	Total Time (s)	Inference Energy (W)	Memory Util. (MB)
Raspberry					
KNN	97.03 [95.61, 98.45]	1.46 [1.03, 1.89]	4.16 [3.73, 4.59]	2.13 [1.30, 2.96]	28
SVM	96.81 [95.39, 98.23]	0.36 [0.17, 0.55]	3.17 [2.74, 3.60]	3.81 [2.97, 4.65]	50
DT	87.8 [86.02, 89.58]	0.12 [0.09, 0.15]	1.58 [1.13, 2.03]	2.32 [1.49, 3.15]	34
Linear	89.55 [87.71, 91.39]	0.21 [0.17, 0.25]	1.0 [0.98, 1.02]	2.71 [2.47, 2.95]	24
Raspberry + Stick					
KNN	97.18 [95.37, 98.99]	1.8 [1.12, 2.48]	4.23 [3.68, 4.78]	2.16 [1.94, 2.38]	33
SVM	96.9 [95.09, 98.71]	0.43 [0.14, 0.72]	3.14 [2.60, 3.68]	3.97 [3.40, 4.54]	52
DT	87.82 [85.95, 89.69]	0.14 [0.12, 0.16]	2.0 [1.59, 2.41]	2.56 [2.10, 3.02]	45
Linear	89.55 [87.83, 91.27]	0.33 [0.23, 0.43]	1.0 [0.98, 1.02]	2.83 [2.48, 3.18]	28
Nvidia Jetson Nano (lite model)					
KNN	97.25 [95.86, 98.64]	0.9 [0.62, 1.18]	3.1 [2.67, 3.53]	3.75 [3.01, 4.49]	35
SVM	97.4 [95.92, 98.88]	0.37 [0.18, 0.56]	2.67 [2.23, 3.11]	4.26 [3.59, 4.93]	68
DT	87.84 [86.02, 89.66]	0.12 [0.10, 0.14]	2.12 [1.74, 2.47]	3.94 [3.38, 4.50]	53
Linear	89.55 [87.71, 91.39]	0.18 [0.12, 0.24]	1.12 [0.97, 1.27]	2.87 [2.62, 3.12]	35
Nvidia Jetson Nano (Tensor-RT model)					
KNN	97.28 [95.86, 98.70]	0.78 [0.64, 0.92]	3.02 [2.58, 3.46]	3.55 [2.91, 4.19]	33
SVM	97.47 [95.92, 98.90]	0.23 [0.11, 0.35]	2.84 [2.39, 3.29]	4.43 [3.88, 4.98]	65
DT	87.88 [86.02, 89.74]	0.12 [0.10, 0.14]	2.1 [1.73, 2.47]	3.93 [3.36, 4.50]	44
Linear	89.57 [87.71, 91.43]	0.18 [0.13, 0.23]	1.14 [0.99, 1.29]	2.81 [2.57, 3.05]	33

shows competitive accuracy when running an FFNN. Its lower memory utilization and power consumption make it a viable option for resource-constrained environments, though it struggles with more demanding tasks like running an R-CNN. Intel Neural Stick, while effective in enhancing the Raspberry Pi's performance, does not match Jetson Nano's capabilities. For instance, when augmenting the Raspberry Pi for tasks using an SVM, the Stick provides marginal improvements in inference time but outpaced by the Jetson Nano's dedicated GPU when handling more complex tasks like running a CNN.

These performance differences are largely due to the specific hardware optimizations in each device. Jetson Nano's GPU excels at handling parallel computations, which are crucial for tasks like running a CNN. In contrast, Raspberry Pi's general-purpose CPU, though energy-eficient, takes longer to process such tasks, making it more suitable for lighter workloads, such as running an FFNN. Intel Stick, designed to supplement CPU-based systems, offers some acceleration but cannot match the Jetson Nano's dedicated GPU performance when running compute-intensive models like R-CNN.

Table 5.9 shows the obtained results for the considered additional deep learning models. As shown in table, the performance analysis of Resnet-50 and MobileSSD

Table 5.8: Performance of neural network models. Metrics are reported along with their confidence interval in square brackets.

Model	Mean Accuracy (%)	Inference Time (s)	Total Time (s)	Inference Energy (W)	Memory Util. (MB)
			Raspberry		
ANN	88.12 [86.89, 89.35]	0.32 [0.31, 0.33]	2.15 [1.97, 2.03]	2.1 [2.08, 2.12]	56
CNN	99.02 [97.79, 100.25]	0.7 [0.68, 0.72]	5.3 [5.21, 5.39]	3.4 [3.34, 3.46]	117
FFNN	97.48 [96.22, 98.68]	0.37 [0.36, 0.38]	3.12 [3.05, 3.19]	2.96 [2.84, 2.96]	98
R-CNN	97.00 [95.77, 98.23]	1.7 [1.66, 1.74]	6.35 [6.26, 6.44]	4.78 [4.71, 4.89]	211
			Raspberry + Stick		
ANN	89.35 [88.11, 90.57]	0.38 [0.37, 0.39]	2.6 [2.55, 2.65]	2.13 [2.10, 2.16]	58
CNN	99.00 [97.77, 100.23]	0.79 [0.77, 0.81]	4.8 [4.75, 4.85]	3.89 [3.81, 3.97]	134
FFNN	97.43 [96.20, 98.66]	0.43 [0.42, 0.44]	3.12 [3.05, 3.19]	3.35 [3.30, 3.40]	93
R-CNN	99.00 [97.77, 100.23]	2.1 [2.07, 2.13]	6.1 [6.02, 6.18]	5.3 [5.24, 5.36]	238
			Nvidia Jetson Nano (lite model)		
ANN	89.13 [87.88, 90.34]	0.31 [0.30, 0.32]	2.33 [2.26, 2.34]	3.2 [3.15, 3.25]	58
CNN	98.48 [97.22, 99.68]	0.82 [0.80, 0.84]	4.3 [4.24, 4.36]	4.67 [4.54, 4.72]	128
FFNN	97.56 [96.33, 98.79]	0.41 [0.40, 0.42]	3.15 [3.09, 3.21]	3.95 [3.86, 3.98]	95
R-CNN	98.74 [97.51, 99.97]	1.1 [1.08, 1.12]	3.9 [3.84, 3.96]	5.79 [5.72, 5.86]	241
			Nvidia Jetson Nano (Tensor-RT model)		
ANN	89.29 [88.07, 90.51]	0.29 [0.28, 0.30]	2.28 [2.22, 2.34]	3.18 [3.14, 3.22]	57
CNN	98.56 [97.28, 99.84]	0.76 [0.74, 0.78]	4.0 [3.95, 4.05]	4.59 [4.48, 4.70]	126
FFNN	97.69 [96.40, 98.98]	0.38 [0.37, 0.39]	3.1 [3.04, 3.16]	3.88 [3.79, 3.97]	93
R-CNN	98.80 [97.55, 100.05]	0.98 [0.96, 1.00]	3.7 [3.64, 3.76]	5.61 [5.54, 5.68]	230

models across various hardware platforms highlights several key observations. As expected, Nvidia Jetson Nano with the TensorRT model demonstrates the best performance. Also, it excels in inference time and maintains eficient memory usage and energy consumption for both models. This advantage is attributed to TensorRT's optimizations, which accelerate model inference through hardware-specific enhancements. For Resnet-50, TensorRT's optimizations significantly reduce inference time and memory usage compared to the other two devices. Similarly, the MobileSSD model benefits from TensorRT's eficiency, achieving faster inference time and competitive accuracy. The Jetson Nano Lite model also performs well but shows slightly higher inference times and energy consumption compared to the TensorRT variant. In contrast, the Raspberry Pi struggles with the MobileSSD model's demands, resulting in longer inference times and higher energy consumption. The Intel Neural Compute Stick improves performance when paired with the Raspberry Pi, which does not match Jetson Nano's eficiency due to limitations in its processing power and optimization for deep models.

We observe that though inference time taken by devices is reducing across the devices in the table, we do not see corresponding change in the total time. Even in some cases, the total time has increased while the inference time is reduced. This aspect can be explained by different phases of model loading and inference, which is measured by total time. Though we observe reduced inference time, some models take longer to load on specific devices and hence we see increased total time.

Table 5.9: Performance of Resnet-50 and MobileSSD models. Metrics are reported along with their confidence interval in square brackets.

Model	Mean Accuracy (%)	Inference Time (s)	Total Time (s)	Inference Energy (W)	Memory Util. (MB)
Raspberry					
Resnet-50	94.76 [93.53, 96.00]	0.57 [0.49, 0.65]	2.8 [2.77, 2.83]	3.3 [3.27, 3.33]	98
MobileSSD	93.78 [92.55, 95.01]	0.42 [0.38, 0.46]	2.7 [2.66, 2.74]	4.7 [4.45, 4.55]	43.21
Raspberry + Stick					
Resnet-50	96.12 [94.89, 97.35]	0.35 [0.27, 0.43]	3.1 [3.03, 3.17]	2.8 [2.74, 2.86]	87
MobileSSD	95.31 [94.10, 96.56]	0.32 [0.28, 0.36]	2.3 [2.24, 2.36]	4.23 [4.15, 4.25]	37.42
Nvidia Jetson Nano (lite model)					
Resnet-50	96.38 [95.15, 97.61]	0.27 [0.20, 0.34]	3.6 [3.54, 3.66]	3.5 [3.45, 3.55]	83
MobileSSD	96.10 [94.96, 97.42]	0.26 [0.23, 0.29]	2.1 [2.07, 2.13]	4.76 [4.62, 4.78]	48
Nvidia Jetson Nano (Tensor-RT model)					
Resnet-50	95.71 [94.48, 96.94]	0.15 [0.11, 0.19]	2.6 [2.55, 2.65]	3.61 [2.54, 2.68]	71
MobileSSD	93.53 [92.30, 94.76]	0.19 [0.16, 0.22]	2.1 [2.06, 2.14]	4.96 [3.85, 4.03]	48

TensorRT's specialized optimizations, such as layer fusion, precision calibration, and kernel tuning, enhance computational eficiency and speed up inference. The Jetson Nano leverages these TensorRT benefits with its powerful GPU architecture, which supports high-performance, parallel processing. In contrast, the Raspberry Pi and Intel Neural Stick, while capable, lack the same level of optimization and specialized hardware support, leading to slower inference times and higher energy consumption. This combination of advanced optimizations and robust hardware makes TensorRT and the Jetson Nano more effective for deploying complex deep learning models.

5.3.1.3 Large Language Models

The performance trend of LLMs is similar to that of other models, which is presented in Table 5.10. This trend is again attributed to the degree of hardware capacities, i.e., Nvidia Jetson Nano is the best performer. The table 5.10 also shows that Tiny-BERT generally outperforms phi-2-orange across all devices in terms of accuracy, inference time, and memory usage. TinyBERT's lightweight, optimized architecture leads to faster processing and lower memory consumption, particularly on resource-constrained devices such as Raspberry Pi. In contrast, phi-2-orange, with its relatively complex architecture, requires more memory and computational resources, which results in slower performance and higher energy consumption.

TinyBERT consistently shows better performance in terms of accuracy, speed, and energy consumption across all devices, from the Raspberry Pi to the Nvidia Jetson Nano. Its lightweight architecture allows it to process tasks faster and with lower energy requirements, making it particularly effective on devices with limited computational resources. On the other hand, phi-2-orange, while still effective, tends to perform less eficiently, particularly in inference time and energy usage. This can be attributed to its more complex model structure, which demands more processing power and memory. As a result, it struggles on lower-end devices like the Raspberry Pi, where resource limitations become more apparent. The advantage of TinyBERT is even more pronounced on devices that support model optimization techniques, like the Nvidia Jetson Nano with Tensor-RT. Here, TinyBERT's architecture, optimized for such accelerations, significantly enhances its speed and reduces energy consumption, further widening the performance gap with phi-2-orange. Phi-2-orange benefits from hardware acceleration too but not to the same extent, likely due to its higher resource demands. In essence, TinyBERT's design allows it to better adapt to the constraints and capabilities of different edge devices, making it the more versatile and eficient choice across varying environments.

Table 5.10: Performance of LLM models. Metrics are reported along with their confidence interval in square brackets.

Model	Mean Accuracy (%)	Inference Time (s)	Total Time (s)	Inference Energy (W)	Memory Util.
			Raspberry		
TinyBERT	87.00 [85.66, 88.34]	0.58 [0.55, 0.61]	4.2 [3.37, 4.03]	3.1 [2.27, 3.43]	41.53MB
phi-2-orange	85.23 [84.03, 86.43]	1.8 [1.62, 1.98]	4.8 [3.96, 5.64]	3.9 [3.06, 4.74]	2.5GB
			Raspberry + Stick		
TinyBERT	89.00 [87.56, 90.44]	0.41 [0.38, 0.44]	3.0 [2.16, 3.84]	3.7 [2.83, 3.91]	32.15MB
phi-2-orange	87.16 [86.04, 88.28]	0.8 [0.64, 0.96]	2.3 [1.47, 3.13]	2.7 [1.86, 3.54]	2.2GB
			Nvidia Jetson Nano (lite model)		
TinyBERT	89.37 [87.93, 90.81]	0.37 [0.34, 0.40]	3.6 [1.76, 3.44]	3.48 [2.67, 4.29]	35MB
phi-2-orange	87.21 [86.02, 88.40]	0.53 [0.46, 0.60]	2.47 [1.64, 3.30]	5.82 [4.97, 6.67]	2.35GB
			Nvidia Jetson Nano (Tensor-RT model)		
TinyBERT	88.20 [86.76, 89.64]	0.24 [0.21, 0.27]	3.3 [1.49, 3.11]	3.47 [2.63, 4.31]	35MB
phi-2-orange	86.45 [85.25, 87.65]	0.33 [0.30, 0.36]	2.11 [1.40, 2.82]	4.72 [3.88, 5.56]	1.78GB

Summary: Our experiment results revealed that traditional and neural network models show similar accuracy between the Raspberry Pi and Raspberry Pi with the Neural Stick, though the latter has a slightly higher power consumption and longer inference time. Models such as Resnet-50 and MobileSSD benefit from the Neu-ral Stick, achieving improved accuracy, reduced inference time, and lower energy consumption under the adopted experimental settings. Similarly, TinyBERT and phi-2-orange models perform best on the Nvidia Jetson Nano, especially with TensorRT optimizations, highlighting the importance of selecting hardware suitable for specific model types for enhanced edge computing performance.

5.3.2 How do lite frameworks impact the learning outcome?

The evaluation of various models across devices reveals that TensorFlow Lite (TFLite), Intermediate Representation (IR), and TensorRT architectures exhibit distinct optimization strategies and performance impacts.

TensorFlow Lite is optimized for edge devices with techniques like quantization and operator fusion, which enhance speed and eficiency [44] but may not fully leverage the capabilities of more powerful hardware, such as the Nvidia Jetson Nano. IR with OpenVINO provides a flexible hardware abstraction layer that allows for tailored optimizations [55] but generally yields less dramatic performance gain compared to TensorRT. TensorRT is particularly designed for Nvidia GPUs [72], offers advanced optimizations such as layer fusion and precision calibration, significantly boosting inference speed, accuracy, and eficiency on Nvidia Jetson Nano devices.

Specifically, deep learning models Resnet-50 and MobileSSD are greatly benefited using Nvidia Jetson Nano with TensorRT. In contrast, while models over Raspberry Pi may demonstrate lower memory usage and energy consumption, they lag behind in inference speed and accuracy. Models optimized with TensorRT on Nvidia Jetson Nano show a clear advantage in performance metrics, highlighting the benefits of leveraging advanced hardware-specific optimizations.

Summary: TensorFlow Lite, Intermediate Representation (IR), and TensorRT each has unique optimization strategies. TensorFlow Lite enhances speed and efi-ciency for edge devices but may not fully utilize powerful hardware like the Nvidia

Jetson Nano. IR models offer flexible optimizations but typically fall short of TensorRT's performance. TensorRT, tailored for Nvidia GPUs, provides superior inference speed, accuracy, and eficiency, particularly evident in Nvidia Jetson Nano performance. While TensorRT-equipped Jetson Nano models outperform Raspberry Pi in speed and accuracy, they often consume more memory and energy. This highlights the need to select model architectures and optimization techniques based on specific performance and eficiency requirements.

5.3.3 What are the trade-offs between performance and resource usage?

This section shows that choosing the optimal combination of machine learning models, hardware devices, and their platforms requires a careful assessment of trade-offs to meet application demand. This analysis explores trade-offs using key performance metrics such as accuracy, inference time, memory usage, and energy consumption.

Table 5.11 shows the performance comparison across edge devices for all models. Traditional models (KNN, SVM, DT, and Linear Regression) can be deployed on Raspberry Pi if applications are not performance intensive; otherwise, Jetson Nano is the choice. The same device is also the best fit for neural network models as it is specifically designed for such models. Due to the same reason, we see Resnet-50 and MobileSSD perform the best across all metrics on Jetson Nano. TinyBERT and phi-2-orange models show interesting performance trends. Both have the best performance on Jetson Nano, but their resource usage is different on different devices, which may require further investigation.

Table 5.11: Performance comparison across edge devices.

Model	Accuracy[%]	Inference Time[S]	Memory Usage[MB]	Energy Consumption[W]
KNN	Jetson Nano	Jetson Nano	Raspberry Pi	Raspberry Pi
SVM	Jetson Nano	Jetson Nano	Raspberry Pi	Raspberry Pi
DT	Tie	Tie	Raspberry Pi	Raspberry Pi
Linear Regression	Tie	Jetson Nano	Raspberry Pi	Raspberry Pi
ANN	Jetson Nano	Jetson Nano	Raspberry Pi	Raspberry Pi
CNN	Raspberry Pi	Jetson Nano	Raspberry Pi	Raspberry Pi
FFNN	Jetson Nano	Jetson Nano	Jetson Nano	Jetson Nano
R-CNN	Jetson Nano	Jetson Nano	Jetson Nano	Jetson Nano
Resnet-50	Jetson Nano	Jetson Nano	Jetson Nano	Jetson Nano
MobileSSD	Jetson Nano	Jetson Nano	Raspberry Pi	Jetson Nano
TinyBERT	Jetson Nano	Jetson Nano	Jetson Nano	Tie
phi-2-orange	Jetson Nano	Jetson Nano	Raspberry Pi	Neural stick

Summary: The Nvidia Jetson Nano consistently outperforms Raspberry Pi in terms of processing speed for most machine learning models, particularly models like Resnet-50, MobileSSD, TinyBERT, and phi-2-orange. This improvement in speed varies from small (ANN, FFNN) to significant (R-CNN, MobileSSD, phi-2-orange). In terms of accuracy, the Nvidia Jetson Nano also demonstrates an edge for the majority of the models tested, except for CNN and Linear models. While the accuracy improvement was marginal for some models (KNN, SVM, DT), it was substantial for others (Resnet-50, MobileSSD, TinyBERT, phi-2-orange). However, it is essential to note that these gains in speed and accuracy on the Nvidia Jetson Nano come at the cost of higher memory usage and energy consumption in some cases (SVM, CNN, Linear model), whereas other models (KNN, ANN, FFNN, Resnet-50, MobileSSD, phi-2-orange) show better eficient resource utilization. Overall, these findings underscore the importance of selecting the appropriate hardware platform depending on the specific requirements of the machine learning model being deployed, balancing factors such as speed, accuracy, memory usage, and energy consumption.

5.4 Key Takeaways

Our experiments highlight the impact of machine learning framework, model architecture, and hardware devices on inference accuracy and computational eficiency. The final takeaways are listed below.

- ML and neural network models show consistent accuracy between Raspberry Pi and Raspberry Pi with Neural Stick, with the latter consuming slightly more power and time for inference.

- Resnet-50 and MobileSSD models demonstrate improved accuracy with the Neural Stick, coupled with reduced inference time and energy consumption.

- The choice of hardware significantly influence performance metrics; while Raspberry Pi variants maintain competitive accuracy, optimizations like TensorRT on Jetson Nano enhance eficiency, especially for resource-intensive models.

Understanding these results is crucial for optimizing machine learning applications on edge devices. The Nvidia Jetson Nano’s superior performance makes it suitable for applications requiring high accuracy and speed, such as real-time object detection or complex model inference. However, its higher resource consumption might not be ideal for battery-powered or memory-constrained devices. On the other hand, the Raspberry Pi provides a more resource-eficient solution for simpler models, making it suitable for applications with limited computational demands or where power consumption is a critical factor. These insights help in making informed decisions about hardware and model selection to achieve the desired balance between performance and resource eficiency.

Chapter 6

Conclusions and Future Work

6.1 Future Work

6.1.1 Potential Future Extensions on IaC Reproducibility

While Reduse has demonstrated eficacy in detecting reproducibility smells in Ansible playbooks, several avenues for future research and tool enhancement exist. For instance, expanding the scope of reproducibility analysis to other IaC frameworks like Terraform, Chef, and Puppet would offer comparative insights into reproducibility challenges across different toolsets.

Furthermore, enhancing Reduse's detection capabilities through machine learning models could enable more sophisticated analysis of complex reproducibility patterns. Integrating Reduse with popular IDEs and code editors used in Ansible development would provide real-time feedback to developers, facilitating immediate remediation of identified smells during playbook creation.

Moreover, refining Reduse to prioritize detected smells based on severity and providing actionable guidance on remediation strategies would further enhance its usability and effectiveness in real-world development scenarios.

6.1.2 Future Extensions of AI Inference on Edge Study

Future work on sustainable AI inference on edge devices could expand in several directions to deepen the understanding and improve the eficacy of AI deployment in resource-constrained environments. First, we can revisit the list of ML models, especially the DL models and LLMs, along with the lightweight frameworks (e.g., PyTorch Mobile) for an extensive evaluation across various models and platforms. In the case of learning models, we see a surge in complex models and their usage. For example, Graph Neural Networks and Transformers are being adopted in various edge applications [16, 21, 47, 48, 57]. We plan to extend the current evaluation to include complex models for their potential usage in the edge. We also can incorporate additional hardware devices like smartNIC and Google Coral. SmartNICs being assessed and deployed in various applications (e.g., load balancing and security) [96, 97, 106, 108]. We plan to replicate the current evaluation on this edge device. Similarly, Google Coral is gaining its attention in edge usage, which we will explore further [79].

6.2 Limitations

6.2.1 Limitations of IaC Reproducibility Study

The heuristics and rules used by Reduse to detect reproducibility smells may introduce construct validity concerns, as they rely on predefined patterns and assumptions about best practices in Ansible playbook development. We mitigate this threat by validating these heuristics through expert evaluations and making Reduse's source code available for public scrutiny and improvement [9]. Our study's external valid-ity may be limited as it focuses exclusively on Ansible as the target IaC framework. While our reproducibility smell catalog is framework-agnostic, future research could expand its applicability to other imperative IaC frameworks to generalize findings across the broader infrastructure management landscape. Internal validity concerns ensuring the reliability and consistency of our study's findings and claims. We address this by employing rigorous research methods, including systematic literature review and empirical analysis of real-world Ansible repositories, to substantiate our findings on reproducibility issues in Ansible playbooks.

6.2.2 Limitations of AI inference Study on Edge

First, we need to extensively evaluate the existing models multiple times for a statistically significant result. Also, the timestamps-based energy measurement can be error-prone, so having an automated system would be better. For example, overlapping or misidentifying phases can lead to inaccuracies in separating each step. We may develop an automated configuration tool as developed in [20], where users can provide their high-level intents of measurements, which the tool can convert for a measurement configuration and visualization of the outcome.

6.3 Conclusions

This thesis addressed critical challenges in reproducibility within Infrastructure as Code and explored sustainable AI inference on edge devices, focusing on practical solutions and future research directions.

In the IaC reproducibility research, we investigated reproducibility challenges in Ansible scripts, a prominent IaC tool. We introduced Reduse, a detection tool for reproducibility smells, which helps practitioners identify and mitigate issues before they impact production systems. Our empirical findings highlighted broken dependency chain as the most prevalent reproducibility smell in analyzed open-source projects, underscoring the necessity of early refactoring. We also provided a comprehensive catalog of reproducibility smells, offering a practical framework for software engineering researchers to explore various dimensions of reproducibility in IaC. All artifacts, including code, testing scripts, and results, are publicly accessible through our replication package [9], promoting the development of reproducible environments. Future improvements to Reduse include expanding the detection framework to cover additional smell categories and extending the study to other IaC tools such as Terraform, Chef, and Puppet.

In the sustainable AI inference research, we evaluated the performance of various machine learning and deep learning models on different edge devices using TensorFlow Lite. Our results demonstrated that hardware choice and model architecture significantly impact performance metrics such as inference time, accuracy, and energy consumption. The Nvidia Jetson Nano, particularly with TensorRT optimizations, emerged as the most eficient platform for edge AI applications, providing superior performance in both speed and energy eficiency. Future work could investigate the integration of other hardware accelerators and further optimization of model architectures to enhance edge AI performance.

By applying the IaC reproducibility research, including the developed tool for detecting reproducibility smells, we can ensure that the IaC scripts used to deploy AI models at the edge are reliable and consistent. This will lead to more reliable deployments of AI models on edge devices, bridging the gap between reproducibility in IaC and sustainable AI inference on edge devices. Overall, this thesis not only addresses immediate challenges in reproducibility within IaC but also lays the groundwork for ongoing advancements in tooling, methodologies, and empirical studies aimed at improving the reliability, eficiency, and sustainability of modern infrastructure management and edge AI inference.

Bibliography

[1] Ansible Galaxy. https://galaxy.ansible.com/home, 2023. Last accessed: July 2024.

[2] Oracle Cloud Infrastructure Ansible Collection. https://github.com/oracle/oci-ansible-collection, 2023.

[3] about neural stick. https://www.intel.com/content/www/us/en/developer/articles/tool/neural-compute-stick.html, 2024.

[4] about nvidia jetson nano. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/, 2024.

[5] about nvidia smi profiling tool. https://developer.nvidia.com/system-management-interface, 2024.

[6] about perf profiling tool. https://firefox-source-docs.mozilla.org/performance/perf.html, 2024.

[7] about raspberry pi. https://www.raspberrypi.com/documentation/computers/getting-started.html, 2024.

[8] Norah N. Alajlan and Dina M. Ibrahim. Tinyml: Enabling of inference deep learning models on ultra-low-power iot edge devices for ai applications. Micromachines, 13(6), 2022.

[9] anonymous. Reduce - replication package, July 2024.

[10] Ansible. Roles Automation and Infrastructure. Last accessed: Jun 18, 2024.

[11] Ansible. Vaults Automation and Infrastructure. Last accessed: Jun 18, 2024.

[12] Ansible. Ansible Best Practices Guide, 2018. Last accessed: July 2024.

[13] Ansible. Ansible Documentation. https://docs.ansible.com/ansible/latest/index.html, 2023. Last accessed: July 2024.

[14] Ansible. Ansible documentation—ad-hoc commands, 2024.

[15] Example—Assumption about environment smell. https://github.com/chocolatey/chocolatey-ansible/blob/9bdc0d40437a7dc7f0181af42da7e35bbcfcae4a/chocolatey/tests/integration/targets/win_chocolatey/tasks/bootstrap_tests.yml#L4, 2023. Last accessed: July 2024.

[16] Tahajjat Begum, Israat Haque, and Vlado Keselj. Deep learning models for gesture-controlled drone operation. In 2020 16th International Conference on Network and Service Management (CNSM), pages 1–7, 2020.

[17] SWAPNIL BHARTIYA. Importance Of Repeatability In IaC — Scaling Infrastructure as Code. https://tfir.io/importance-of-repeatability-in-iac-scaling-infrastructure-as-code/, 2022.

[18] Farzana Ahamed Bhuiyan and Akond Rahman. Characterizing Co-located Insecure Coding Patterns in Infrastructure as Code Scripts. In 2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), pages 27–32, 2020.

[19] Bluelight. Best Infrastructure as Code (IaC) Tools, 2023. Last accessed: July 2024.

[20] Conrado Boeira et al. Calibration and automation of a 5G simulator for realistic evaluation and data generation. In accepted in IEEE Conference on Network Softwarization (NetSoft). IEEE, 2024.

[21] Conrado Boeira, Antor Hasan, Khaleda Papry, Yue Ju, Zhongwen Zhu, and Israat Haque. A calibrated and automated simulator for innovations in 5G, 2024.

[22] Exmaple—Broken dependency chain smell. https://gitlab.com/idris-cnrs/jupyter/ansible-jupyterhub-hpc/-/blob/main/roles/setup_jupyterhub/tasks/tokens.yml?ref_type=heads, 2023. Last accessed: July 2024.

[23] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. Proceedings of the IEEE, 107(8):1655–1674, 2019.

[24] Michele Chiari, Michele De Pascalis, and Matteo Pradella. Static Analysis of Infrastructure as Code: a Survey. In 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C), pages 218–225, 2022.

[25] cisco project for qa. https://github.com/ansible-collections/cisco.nxos, 2024.

[26] Cisco asa. https://github.com/ansible-collections/cisco.asa/issues/195, 2024.

[27] Cisco asa. https://github.com/ansible-collections/cisco.asa/issues/196, 2024.

[28] Christian Collberg and Todd A. Proebsting. Repeatability in Computer Systems Research. Commun. ACM, 59(3):62–69, 2016.

[29] Ting Dai, Alexei Karve, Grzegorz Koper, and Sai Zeng. Automatically detecting risky scripts in infrastructure code. pages 358–371, 2020.

[30] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. Toward a catalog of software quality metrics for infrastructure code. Journal of Systems and Software, 170:110726, 2020.

[31] Tamas Das. Infrastructure as Code vs Configuration Management, 2022. Last accessed: July 2024.

[32] Example—Outdated dependency smell. https://github.com/cchurch/ansible-role-virtualenv/blob/master/tasks/update.yml, 2023. Last accessed: July 2024.

[33] EdgeAI. https://gitlab.com/sobhanii/edgeai, 2024.

[34] Dimma Enns. Troubleshooting Dependency Version Conflict, 2021. Last accessed: July 2024.

[35] Roy Feintuch. New Security Challenges with Infrastructure as Code and Immutable Infrastructure, 2018. Last accessed: July 2024.

[36] Dror G. Feitelson. From Repeatability to Reproducibility and Corroboration. SIGOPS Oper. Syst. Rev., 49(1):3–11, 2015.

[37] Marco Ferrari. Want Repeatable Scale? Adopt Infrastructure as Code on GCP, 2020. Last accessed: July 2024.

[38] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Information and Software Technology, 106:101–121, 2019.

[39] Vahid Garousi and Barış Küçük. Smells in software test code: A survey of knowledge in industry and academia. Journal of systems and software, 138:52–81, 2018.

[40] Stefanos Georgiou, Maria Kechagia, Tushar Sharma, Federica Sarro, and Ying Zou. Green AI: Do Deep Learning Frameworks Have Different Costs? In Proceedings of the 44th International Conference on Software Engineering, ICSE '22, page 1082–1094, 2022.

[41] Ananda M. Ghosh and Katarina Grolinger. Deep learning: Edge-cloud data analytics for iot. In 2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE), pages 1–7, 2019.

[42] GitLab. Upgrading Auto Deploy Dependencies, 2022.

[43] GitLab. Infrastructure as Code (IaC) Scanning, 2023. Last accessed: July 2024.

[44] google. about the tensorflow lite framework. https://www.tensorflow.org/lite, 2024.

[45] Tiexin Guo. Managing Infrastructure with Terraform, 2021. Last accessed: July 2024.

[46] Example—Hardware specific command smell. https://github.com/aplatform64/aplatform64/blob/3d563246263f6d2a83de604296704a4b76164caa/docs/examples/hw_gpu_amd.yml#L4, 2023. Last accessed: July 2024.

[47] Antor Hasan, Conrado Boeira, Khaleda Papry, Yue Ju, Zhongwen Zhu, and Israat Haque. NetRepAIr - making networks reliable for next-generation applications using AI/ML techniques, 2024.

[48] Kazi Hasan, Thomas Trappenberg, and Israat Haque. A generalized transformer-based radio link failure prediction framework in 5G RANs, 2024.

[49] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Testing idempotence for infrastructure as code. In David Eyers and Karsten Schwan, editors, Middleware 2013, pages 368–388, 2013.

[50] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Testing idempotence for infrastructure as code. In David Eyers and Karsten Schwan, editors, Middleware 2013, pages 368–388, 2013.

[51] Snyk IaC. Infrastructure as Code Security, 2023. Last accessed: July 2024.

[52] Example—Unguarded operation smell. https://github.com/openshift/openshift-ansible/blob/master/roles/openshift_node/tasks/apply_machine_config.yml, 2023. Last accessed: July 2024.

[53] imagenet. https://www.tensorflow.org/datasets/catalog/imagenet2012, 2024.

[54] InfinityPP. Ansible Best Practices, 2020. Last accessed: July 2024.

[55] Intel. about the intel ir framework. https://docs.openvino.ai/2023.3/openvino_ir.html, 2024.

[56] Chadni Islam, Muhammad Ali Babar, and Surya Nepal. A multi-vocal review of security orchestration. ACM Computing Surveys (CSUR), 52(2):1–45, 2019.

[57] Mohammad Ariful Islam, Hisham Siddique, Wenbin Zhang, and Israat Haque. A deep neural network-based communication failure prediction scheme in 5g ran. IEEE Transactions on Network and Service Management, 2022.

[58] Example—incompatible version dependency smell. https://github.com/geerlingguy/ansible-for-devops/blob/master/includes/provisioning/tasks/common.yml, 2023. Last accessed: July 2024.

[59] Shahedul Huq Khandkar. Open coding. https://pages.cpsc.ucalgary.ca/~saul/wiki/uploads/CPSC681/open-coding.pdf, 2009.

[60] Indika Kumara, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. The Do's and Don'ts of Infrastructure Code: A Systematic Gray Literature Review. Information and Software Technology, 137:106593, 2021.

[61] Indika Kumara, Zoe Vasileiou, Georgios Meditskos, Damian A. Tamburri, Willem-Jan Van Den Heuvel, Anastasios Karakostas, Stefanos Vrochidis, and Ioannis Kompatsiaris. Towards Semantic Detection of Smells in Cloud Infrastructure Code. 2020.

[62] Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi-Rui Tam, Keith Stevens, Abdullah Barhoum, Nguyen Minh Duc, Oliver Stanley, Richárd Nagyfi, Shahul ES, Sameer Suri, David Glushkov, Arnav Dantuluri, Andrew Maguire, Christoph Schuhmann, Huu Nguyen, and Alexander Mattick. Openassistant conversations – democratizing large language model alignment, 2023.

[63] Ricardo Matsui. Stop Using Peer Dependencies, 2018. Last accessed: July 2024.

[64] Medium. Infrastructure as Code at Tile: Benefits, Advantages, and Features, 2018. Last accessed: July 2024.

[65] medium. On DevOps #8 - Infrastructure as Code Introduction, Best Practices, and Choosing the Right Tool, 2021. Last accessed: July 2024.

[66] Microsoft. Automation and Infrastructure, 2023. Last accessed: July 2024.

[67] mnist. https://www.tensorflow.org/datasets/catalog/mnist, 2024.

[68] Kief Morris. Infrastructure as Code: Managing Servers in the Cloud. 2016.

[69] Kief Morris. Infrastructure as code. O'Reilly Media, 2020.

[70] Wahl Network. Dependency Pinning with Infrastructure as Code, 2020. Last accessed: July 2024.

[71] Novaordis. Infrastructure as Code Concepts, 2021. Last accessed: July 2024.

[72] Nvidia. about the tensorrt framework. https://developer.nvidia.com/tensorrt#section-what-is-nvidia-tensorrt, 2024.

[73] about the onnx framework. https://onnxruntime.ai/, 2024.

[74] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. Andromeda: A Dataset of Ansible Galaxy Roles and Their Evolution. In 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pages 580–584, 2021.

[75] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime. pages 61–72, 2022.

[76] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort? In IEEE/ACM 20th International Conference on Mining Software Repositories (MSR 2023), 2023.

[77] about the pytorch mobile framework. https://pytorch.org/mobile/home/#key-features, 2024.

[78] Xuan Qi and Chen Liu. Enabling deep learning on iot edge: Approaches and evaluation. In 2018 IEEE/ACM Symposium on Edge Computing (SEC), pages 367–372, 2018.

[79] Tobiasz Rafal, Wilczynski Grzegorz, Graszka Piotr, Czechowski Nikodem, and Luczak Sebastian. Edge devices inference performance comparison. Journal of Computing Science and Engineering, 17(2):51–59, June 2023.

[80] A. Rahman, E. Farhana, C. Parnin, and L. Williams. Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts. pages 752–764, 2020.

[81] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. A systematic mapping study of infrastructure as code research. Information and Software Technology, 108:65–77, 2019.

[82] Akond Rahman and Tushar Sharma. Lessons from Research to Practice on Writing Better Quality Puppet Scripts. In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 63–67, 2022.

[83] Akond Rahman and Laurie Williams. Source code properties of defective infrastructure as code scripts. Information and Software Technology, 112:148–163, 2019.

[84] Mohammad Wali Ur Rahman. Optimizing large language models for edge devices: A comparative study on reputation analysis, 2023.

[85] Gonzalo Rivero and Jiating (Kristin) Chen. Best coding practices to ensure reproducibility. 2020.

[86] Nuno Saavedra and João F. Ferreira. GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE '22, 2023.

[87] Julian Schwarz, Andreas Steffens, and Horst Lichter. Code Smells in Infrastructure as Code. In 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), pages 220–228, 2018.

[88] S. S. SHAPIRO and M. B. WILK. An analysis of variance test for normality (complete samples). Biometrika, 52(3-4):591–611, 1965.

[89] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does Your Configuration Code Smell? In 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pages 189–200, 2016.

[90] Tushar Sharma and Diomidis Spinellis. A survey on software smells. Journal of Systems and Software, 138:158 – 173, 2018.

[91] Snyk. Infrastructure as Code (IaC) - A Comprehensive Guide, 2023. Last accessed: July 2024.

[92] BMC Software. Infrastructure as Code: Definitions, Advantages, Best Practices, and Features, 2020. Last accessed: July 2024.

[93] OTEEMO Software. Organizing Ansible, 2018. Last accessed: July 2024.

[94] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. Journal of Systems and Software, 146:215–232, 2018.

[95] Online Questionnaire for IaC experts. https://forms.office.com/r/s27JTNDtrC, 2024.

[96] Hesam Tajbakhsh, Ricardo Parizotto, Miguel Neves, Alberto Schaeffer-Filho, and Israat Haque. Accelerator-aware in-network load balancing for improved application performance. In 2022 IFIP Networking Conference (IFIP Networking), pages 1–9, 2022.

[97] Hesam Tajbakhsh, Ricardo Parizotto, Alberto Schaeffer-Filho, and Israat Haque. P4hauler: An accelerator-aware in-network load balancer for applications performance boosting. IEEE Transactions on Cloud Computing, 12(2):697–711, 2024.

[98] TechTarget. Perforce Acquires Puppet for Infrastructure as Code, 2018. Last accessed: July 2024.

[99] Thorntech. Infrastructure as Code Best Practices, 2015. Last accessed: July 2024.

[100] Shusaku Tsumoto and Shoji Hirano. Contingency matrix theory. In 2007 IEEE International Conference on Systems, Man and Cybernetics, pages 3778–3783, 2007.

[101] Roberto Verdecchia, Ivana Malavolta, and Patricia Lago. Guidelines for architecting android apps: A mixed-method empirical study. In 2019 IEEE International Conference on Software Architecture (ICSA), pages 141–150. IEEE, 2019.

[102] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding, 2019.

[103] Fangxin Wang, Miao Zhang, Xiangxiang Wang, Xiaoqiang Ma, and Jiangchuan Liu. Deep learning for edge computing applications: A state-of-the-art survey. IEEE Access, 8:58322–58336, 2020.

[104] R. Wang. Infrastructure as Code, Patterns and Practices: With examples in Python and Terraform. Manning, 2022.

[105] Wikipedia contributors. Phi coeficient — Wikipedia, the free encyclopedia, 2024.

[106] Tong Xing, Hesam Tajbakhsh, Israat Haque, Michio Honda, and Antonio Barbalace. Towards portable end-to-end network performance characterization of smartnics. In Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems, pages 46–52, 2022.

[107] Xingzhou Zhang, Yifan Wang, and Weisong Shi. pcamp: Performance comparison of machine learning packages on the edges. 07 2018.

[108] Jack Zhao, Miguel Neves, and Israat Haque. On the (dis) advantages of programmable nics for network security services. In 2023 IFIP Networking Conference (IFIP Networking), pages 1–9. IEEE, 2023.