Titre: Title:	Sustainability Smells in Infrastructure as Code
Auteur: Author:	Seifeldin Kosbar
Date:	2026
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Kosbar, S. (2026). Sustainability Smells in Infrastructure as Code [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/73177/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Sustainability Smells in Infrastructure as Code

SEIFELDIN KOSBAR
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Février 2026

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Sustainability Smells in Infrastructure as Code

présenté par **Seifeldin KOSBAR**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Zohreh SHARAFI, présidente
Mohammad HAMDAQA, membre et directeur de recherche
Heng LI, membre

DEDICATION

Mount Royal and its city,
you have changed me a lot...

ACKNOWLEDGEMENTS

First of all, I would like to thank **Allah**. All power and knowledge is from him. I always ask him for many things and he gives me more than I ask. Alhamdulillah!

I would like to express my deepest gratitude to my supervisor Prof. Mohammad Hamdaqa for his continuous guidance, invaluable insights, and unwavering support throughout this research journey. His expertise and mentorship have been instrumental in shaping this work, and his dedication to providing every resource necessary has enabled me to grow both as a researcher and as an individual.

I would like to thank the committee members, Professors Zohreh Sharafi and Heng Li, for kindly agreeing to evaluate this work and for the invaluable experience and insights they bring through their jury judgment and feedback.

I am forever grateful to my mother, my father and my sister for their unconditional love, endless encouragement, and unwavering belief in my abilities. Their sacrifices and support throughout my life journey have been the foundation of all my achievements.

To my friends and colleagues, thank you for your companionship, moral support, and for making this challenging two-year journey more enjoyable. Your friendship has supported me through the inevitable ups and downs of graduate research.

RÉSUMÉ

L'Infrastructure as Code (IaC) est au cœur de l'ingénierie cloud moderne, en permettant de provisionner et de gérer l'infrastructure au moyen de fichiers de configuration lisibles par machine. Si l'IaC améliore la reproductibilité et la scalabilité, elle peut également cristalliser des décisions d'infrastructure inefficaces : certains motifs récurrents de configuration peuvent entraîner une consommation de ressources inutile, des coûts opérationnels accrus et un impact environnemental évitable. Pour caractériser ces inefficacités, cette thèse introduit les *odeurs de durabilité* (*sustainability smells*) comme des indicateurs de risque de gaspillage potentiel ou d'utilisation sous-optimale des ressources dans des scripts Terraform.

Cette thèse poursuit deux objectifs : (1) définir et valider empiriquement une taxonomie d'odeurs de durabilité pour Terraform, et (2) mesurer leur prévalence à grande échelle tout en améliorant la fiabilité de l'estimation de prévalence. Nous dérivons sept odeurs, fondées sur les recommandations des fournisseurs cloud et sur les retours de praticiens, puis nous affinons la taxonomie au moyen d'une analyse qualitative d'un échantillon stratifié. Nous étudions ensuite la prévalence sur un large corpus de plus de 28 000 scripts Terraform provenant de centaines de dépôts open source. Les résultats montrent que les odeurs de durabilité sont fréquentes mais inégalement réparties entre les projets, et que les inefficacités apparentes dépendent souvent d'interactions entre la structure des scripts, les choix de configuration et des contraintes contextuelles.

Afin d'améliorer l'estimation de la prévalence au-delà d'heuristiques statiques fondées sur des règles, nous construisons un benchmark annoté et quantifions l'erreur de mesure de règles basées sur des expressions régulières, en mettant en évidence leurs limites pour des odeurs dépendantes du contexte (p. ex., le sur-provisionnement et les inefficacités de transfert de données). Nous proposons ensuite un pipeline d'estimation de prévalence sensible au contexte, qui exploite les informations de contrôle de version en analysant conjointement les diffs et les messages de commit avec des grands modèles de langage (LLM). À travers du few-shot prompting, des études d'ablation et une validation croisée entre modèles, l'approche proposée produit des estimations de prévalence plus stables et plus exactes, et permet une analyse qualitative complémentaire des manières dont les développeurs atténuent les odeurs de durabilité. Enfin, nous synthétisons des stratégies de mitigation récurrentes, notamment le *rightsizing*, l'autoscaling, l'optimisation des chemins réseau, l'affinement du cycle de vie des ressources, l'amélioration de la gestion de l'état et la modularisation.

Au final, cette thèse contribue (1) une taxonomie validée d'odeurs de durabilité pour Ter-

raform, (2) une étude empirique de leur prévalence à l'échelle de l'écosystème, et (3) une approche sensible au contexte visant à améliorer la robustesse de l'estimation de prévalence et à mettre en évidence les pratiques de mitigation des développeurs.

ABSTRACT

Infrastructure as Code (IaC) is central to modern cloud engineering, enabling infrastructure provisioning and management through machine-readable configuration files. While IaC improves reproducibility and scalability, it can also encode inefficient infrastructure decisions: recurring configuration patterns may lead to unnecessary resource consumption, increased operational costs, and avoidable environmental impact. To characterize such inefficiencies, this thesis introduces *sustainability smells* as risk indicators of potential waste or suboptimal resource usage in Terraform scripts.

This thesis has two goals: (1) to define and empirically validate a taxonomy of sustainability smells for Terraform, and (2) to measure their prevalence at scale while improving the reliability of prevalence estimation. We derive seven smells grounded in cloud provider guidance and practitioner feedback, and we refine the taxonomy through qualitative analysis of a stratified sample. We then study prevalence on a large corpus of **28,327 Terraform scripts** from **386** open-source repositories. The results show that sustainability smells are frequent but unevenly distributed across projects, and that apparent inefficiencies often depend on interactions between script structure, configuration choices, and contextual constraints.

To improve prevalence estimation beyond static, rule-based heuristics, we construct a labeled benchmark of **148 repositories** and quantify the measurement error of regex-based rules, highlighting their limitations for context-dependent smells (e.g., over-provisioning and data-transfer inefficiencies). We then propose a context-aware prevalence estimation pipeline that leverages version-control context by jointly analyzing commit diffs and messages with large language models (LLMs). Through few-shot prompting, ablation studies, and cross-model validation, the proposed approach yields consistently lower absolute deviation than regex baselines; this result is supported by manual review of sampled cases and enables a complementary qualitative analysis of how developers mitigate sustainability smells. Finally, we synthesize recurring mitigation strategies, including rightsizing, autoscaling, network-path optimization, lifecycle refinement, state-management improvements, and modularization.

Overall, this thesis contributes (1) a validated sustainability-smell taxonomy for Terraform, (2) an empirical prevalence study at ecosystem scale (28,327 scripts from 386 repositories), and (3) a context-aware approach to improve the robustness of prevalence estimation and to surface developer mitigation practices.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

IaC	Infrastructure as Code
AWS	Amazon Web Services
GCP	Google Cloud Platform
Azure	Microsoft Azure
VM	Virtual Machine
LLM	Large Language Model
EC2	Elastic Compute Cloud
S3	Simple Storage Service
GCS	Google Cloud Storage
SS	Sustainability Smell
CI	Continuous Integration
CD	Continuous Deployment

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

Cloud computing has transformed the way modern systems are deployed and managed, enabling organizations to scale seamlessly while maintaining high availability and operational efficiency. However, this rapid growth has come with significant environmental and economic implications. Figure 1.1 summarizes recent analyses estimating that data centers consume between 240 and 340 TWh of electricity annually, representing roughly 1–1.3% of global energy demand [1]. Forecasts further suggest that by 2030, cloud services could account for up to 40% of global electricity usage and emit up to 5.5% of global carbon emissions [2]. These trends highlight the urgent need for sustainable cloud engineering practices.

Infrastructure as Code (IaC) has become a foundational pillar in modern DevOps pipelines. IaC enables practitioners to define, provision, and manage infrastructure using machine-readable configuration files [3]. A large-scale study reported that 69% of practitioners work with more than three IaC tools, with Terraform being among the most widely used [4]. Terraform supports declarative resource provisioning across major cloud providers including AWS, Azure, and GCP [5–8], allowing developers to manage virtual machines, networks, storage, and security policies programmatically [9, 10].

Despite its benefits, IaC is a form of software and can suffer from defects and suboptimal practices. Such issues can lead to severe operational consequences: for instance, a defective IaC script deleted home directories of 270 Wikimedia Commons users [11], and another misconfiguration caused an AWS outage with losses exceeding $150 million [12]. As with conventional software, recurring negative patterns or "anti-patterns" [13] can emerge in IaC codebases, affecting performance, resilience, or maintainability. Yet, an equally critical dimension has received limited attention: sustainability.

Cloud infrastructures can inadvertently waste resources due to misconfigurations, poor architectural decisions, or inefficient provisioning strategies. Examples include over-provisioning compute resources, generating excessive network traffic, mismanaging state, or creating monolithic deployment architectures [6]. Motivations for cloud adoption include cost-effectiveness and efficient resource usage [14], but these benefits diminish when infrastructure is configured inefficiently. Inefficiencies can also inflate the environmental footprint of cloud applications, increasing carbon emissions and energy demand.

However, empirical research on sustainability issues in IaC and particularly on how these inefficiencies manifest within Terraform scripts remains scarce [15]. Unlike security or maintainability smells, sustainability-related issues have not been formalized or systematically

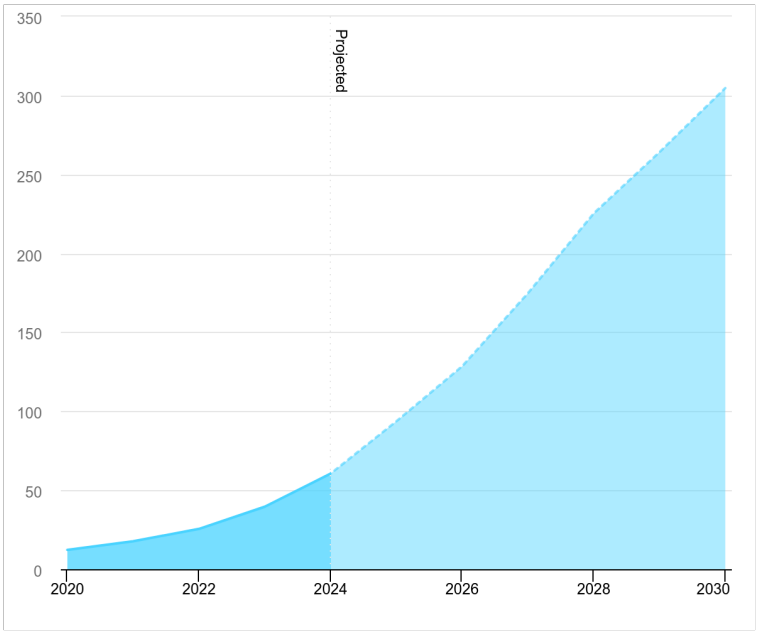

Figure 1.1 Global data center electricity use, 2020–2030 (adapted from IEA data)

studied in the IaC domain. To fill this gap, **this thesis introduces the concept of *sustainability smells***: recurring IaC patterns that serve as risk indicators of cost inefficiency in IaC.

Throughout this thesis, the term sustainability refers specifically to infrastructure-level resource efficiency and cost-related waste observable through static analysis of IaC artifacts, rather than broader environmental or social sustainability concerns.

1.1 Research Goals

As organizations increasingly rely on IaC to automate cloud deployments, they also become responsible for ensuring that their infrastructure is resource-efficient, cost-effective, and environmentally sustainable. However, practitioners currently lack a structured framework to identify sustainability-related issues in IaC, and no tooling exists to detect or mitigate them effectively. This thesis aims to provide the first comprehensive empirical foundation for sustainability analysis in IaC.

The overarching goal of this thesis is to build a comprehensive, data-driven understanding of sustainability issues in IaC including how they manifest in code, how practitioners perceive them, how frequently they occur, and how they are addressed in practice. Within this unified goal, each research question plays a specific and connected role: RQ1 defines sustainability

smells, RQ2 validates their practical relevance, RQ3 measures their prevalence at scale, RQ4 assesses the accuracy of rule-based detection, RQ5 examines whether contextual reasoning improves detection quality, and RQ6 identifies how developers mitigate these issues in real-world settings.

The research is guided by the following questions:

1. **RQ1: What sustainability smells occur in IaC scripts?**
   Derive a grounded taxonomy of sustainability smells by combining (1) a qualitative examination of a stratified sample of Terraform scripts from Dataset1 (1,860 sampled scripts from 395 repositories), (2) alignment with cloud provider sustainability and cost guidance, and (3) iterative refinement through code-based evidence. Success for RQ1 is a concise set of well-defined smell categories with concrete, code-level descriptions and illustrative examples.

2. **RQ2: How do practitioners perceive the proposed sustainability smells?**
   Validate the conceptual relevance, clarity, and practical importance of the derived smells via a targeted survey of experienced IaC practitioners (19 respondents). This question evaluates face and construct validity by collecting practitioners' judgments on smell definitions, perceived severity, and real-world relevance, and captures qualitative feedback used to refine definitions and naming.

3. **RQ3: How frequently do the identified sustainability smells occur in real-world Terraform repositories?**
   Measure prevalence across the full Dataset1 corpus (28,327 Terraform scripts) using operationalized detection rules (initially regex-based) and stratified manual verification. This RQ quantifies per-smell occurrence rates, repository-level distributions, and co-occurrence patterns, and reports confidence intervals and sampling-based error estimates.

4. **RQ4: How accurate are static, rule-based methods at measuring sustainability smells prevalence?**
   Rigorously evaluate regex-based heuristics against ground truth by (1) constructing a labeled benchmark (Dataset2 / curated ground-truth repositories), (2) computing standard detection metrics (precision, recall, F1, and absolute prevalence error), and (3) analyzing failure modes. The goal is to identify which smells are suitable for static detection and which require deeper contextual reasoning.

5. **RQ5: Can commit-level context and reasoning improve the contextual interpretation of sustainability smell prevalence?**
   Develop and evaluate a context-aware detection pipeline that reasons over commit diffs and messages using large language models (LLMs). This RQ covers prompt design, ablation of input modalities (diffs, messages, regex cues), cross-model agreement, and aggregation rules used to produce high-confidence detections. Success is demonstrated through improved detection metrics and more stable prevalence estimates compared to regex-only baselines.

6. **RQ6: What mitigation strategies do developers use to address sustainability smells?**
   Analyze commits flagged by the context-aware pipeline to extract and categorize recurring mitigation actions (e.g., rightsizing, autoscaling, lifecycle rules, modularization). Combine automated LLM-assisted thematic analysis with manual validation to produce a taxonomy of remediation strategies and derive recommendations for automated detection-and-remediation tooling.

1.2 Thesis Contributions

This thesis contributes novel concepts, empirical analyses, and detection methodologies to the emerging domain of sustainable cloud infrastructure engineering. The main contributions are:

- Introduction and validation of sustainability smells as a new class of IaC inefficiencies.
- To our knowledge, the first large-scale empirical study of sustainability smells across 28,327 Terraform scripts.
- An evaluation of regex-based detection and its limitations for context-dependent sustainability issues.
- A context-aware detection framework using LLMs and commit histories.
- A taxonomy of mitigation strategies derived from LLM-flagged commits.
- Datasets, detectors, and prompts enabling replication and future research.

1.3 Thesis Overview and Structure

This thesis is organized as follows:

- **Chapter 1** introduces the research context, goals, and contributions.
- **Chapter 2** provides background on cloud sustainability, IaC, Terraform, and code smell concepts.
- **Chapter 3** presents the derivation and categorization of sustainability smells (RQ1, RQ2).
- **Chapter 4** measures prevalence, introduces the LLM-based detection approach, and analyzes developer mitigation strategies based on the results (RQ3, RQ4, RQ5, RQ6).
- **Chapter 5** concludes the thesis and outlines future work.

CHAPTER 2 BACKGROUND & RELATED WORK

In this chapter, we define foundational concepts for our study and highlight the challenges that motivate it, particularly regarding sustainability standards in IaC.

2.1 Cloud Computing

Cloud computing is a technology paradigm enabling users to access a shared pool of virtualized computing resources such as processing power, storage, networking, and applications via the internet on demand, often with a utility-style, pay-as-you-go model [16, 17]. According to NIST [16], it encompasses five essential characteristics: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service, establishing a clear technical foundation for evaluating cloud systems. ISO/IEC 22123-1:2023 further refines this vocabulary, aligning global terminology and distinguishing among related cloud service and deployment models [17]. The historical roots of cloud computing extend back to the 1960s, with the vision of John McCarthy's "utility computing" and early time-sharing systems such as Project MAC, followed by commercial implementations like AWS's launch of S3 and EC2 in 2006, which operationalized scalable virtual infrastructure for on-demand use [18]. Service models including Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) define the technical layers through which cloud capabilities are delivered, each offering varying degrees of user control and abstraction over infrastructure [19].

2.2 Cloud Deployment Models

Cloud computing resources can be provisioned through various deployment models, each tailored to specific organizational needs. Public clouds offer computing services via shared infrastructure managed by third-party providers, making resources widely accessible, cost-effective, and highly scalable [20]. Private clouds, conversely, are dedicated infrastructures operated exclusively for a single organization, ensuring enhanced control, customization, and data security at the expense of increased complexity and operational overhead [21]. Hybrid cloud environments integrate both public and private clouds, providing flexibility by allowing organizations to leverage the scalability and cost-effectiveness of public resources while preserving sensitive workloads within private infrastructure [22]. Recently, multi-cloud deployments, which involve simultaneously utilizing services from multiple cloud providers,

have gained popularity for their potential to optimize cost, avoid vendor lock-in, and enhance redundancy [23]. Each deployment model significantly influences an organization's resource utilization, management complexity, and sustainability potential.

2.3 Cloud Elasticity and Scalability

A foundational attribute of cloud computing is its inherent elasticity and scalability, enabling dynamic resource allocation in response to workload demands. Elasticity refers specifically to the ability of cloud infrastructures to provision or de-provision resources automatically, closely matching fluctuating application demands [24]. Scalability, on the other hand, describes the capacity of a system to handle increased workload by expanding available resources either vertically, through upgrading existing resources, or horizontally, by adding more resource instances [25]. The effectiveness of elasticity and scalability strategies is essential for maintaining performance, optimizing resource utilization, and achieving operational efficiency. Mismanagement in this domain can lead to resource wastage or service degradation, underscoring the importance of elasticity and scalability considerations within the broader context of sustainable cloud computing.

2.4 Data Centers and Cloud Infrastructure

Data centers serve as the physical backbone of cloud computing, housing extensive networks of servers, storage systems, networking equipment, and cooling infrastructures required for reliable cloud service delivery [26]. These facilities can vary significantly in size, ranging from modest installations to massive hyperscale complexes distributed globally. Such expansive infrastructures necessitate sophisticated management techniques, including resource virtualization, workload consolidation, and adaptive cooling strategies, to maintain performance and operational efficiency [21]. Moreover, the geographic distribution of data centers enables strategic placement close to user populations, which minimizes latency and enhances user experience but also introduces challenges related to infrastructure redundancy, energy use, and resource management [27]. Thus, data center management and infrastructure design remain critical considerations in the sustainability of cloud computing environments.

2.5 Workload Context in Cloud Computing

In cloud environments, resource consumption and energy footprint strongly depend on workload behavior, scaling patterns, time-of-day usage, data transfer volume, and deployment

configurations. Sustainability decisions cannot be made solely based on infrastructure code or resource declarations; they require knowledge of runtime context, workload variability, and scaling behaviors. Recent work highlights the importance of workload-aware scheduling, dynamic resource provisioning, and energy-efficient workload placement to reduce cloud energy consumption [28]. Without workload context, static resource allocations or naïve heuristics often lead to over-provisioning, underutilization, or inefficient resource use. This insight motivates our focus on detecting sustainability smells not just through syntax, but via context including deployment history, scaling events, and developer intent.

2.6 Infrastructure as Code (IaC)

Infrastructure as Code is an approach to managing and provisioning computing infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools [29]. This approach uses high-level scripting languages to automate the setup, configuration, and management of infrastructure components such as servers, networks, and databases. IaC is a key DevOps practice and an essential aspect of continuous delivery [30]. It allows organizations to automate the provisioning process, ensuring that environments are set up consistently every time, and enabling teams to track changes, collaborate, and roll back to previous configurations if necessary [31]. However, while IaC streamlines infrastructure management, it introduces new challenges around sustainability, particularly regarding resource optimization and environmental impact.

2.7 The Role of Terraform in Infrastructure as Code

Terraform is among the most widely adopted Infrastructure as Code (IaC) tools, supporting declarative definition of cloud resources across major providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) [5–8]. Its popularity stems from its provider-agnostic resource model, modular design with reusable components (modules), and support for declarative, idempotent provisioning. Terraform scripts enable developers to define compute instances, storage, networks, permissions, autoscaling, and more in version-controlled configuration files, enabling repeatable and automated infrastructure deployment.

At the same time, the flexibility and power of Terraform introduce sustainability risks: without careful design and upkeep, IaC scripts may encode inefficiencies such as over-provisioning, monolithic resource definitions, unused resources, or poor scaling strategies. Given Terraform's dominant position in IaC ecosystems and open-source prevalence, it serves as an ideal target for our study. The empirical analyses in this thesis are largely based on real-

world Terraform repositories, allowing us to assess sustainability smells in realistic cloud infrastructures.

2.8 Sustainability Definition in IaC Context

This section clarifies the *definition and scope* of sustainability adopted in this thesis. We consider sustainability from a narrow, infrastructure-centric perspective, focusing on resource efficiency and cost-related inefficiencies observable at the IaC level. Accordingly, we define sustainability smells as recurring configuration patterns in IaC scripts that may indicate potential inefficiencies in infrastructure provisioning, such as over-allocation of compute resources, lack of elasticity, or unnecessary data transfer. These inefficiencies are treated as risk indicators rather than direct measurements of environmental impact.

This definition intentionally excludes broader dimensions of sustainability, such as social, organizational, or lifecycle-based environmental impacts, which cannot be reliably inferred from static IaC artifacts alone. While cost and energy consumption are not equivalent, infrastructure cost is commonly used in cloud engineering as a practical proxy for inefficient resource allocation, which often correlates with increased energy usage at the infrastructure layer. Consequently, sustainability smells in this work should be interpreted as heuristics that flag configurations warranting further inspection, not as definitive evidence of unsustainable infrastructure.

2.9 Sustainability Standards in IaC

Although there are industry standards for cloud security, compliance, and efficiency, there is no established framework specifically focused on IaC sustainability [32–34]. While cloud providers like AWS, Azure, and Google Cloud offer guidelines for cloud sustainability, these recommendations vary in scope and focus, leading to fragmented sustainability practices across providers [6–8]. For example, some platforms prioritize energy-efficient data center locations, while others focus on reducing idle resource usage. The lack of uniform sustainability standards for IaC results in an ad hoc approach, where practitioners independently interpret and apply sustainable practices, leading to inconsistency. Our research seeks to address this gap by exploring and proposing structured guidelines for sustainable IaC practices, encouraging more consistent and effective deployment strategies that align with sustainability objectives.

2.10 Git Commits as a Source of Contextual Data

Version control systems and commit histories on platforms such as GitHub provide temporal and contextual information about how software artifacts evolve over time [35, 36]. Commits include diffs, commit messages, and metadata (author, timestamp, files changed); these signals can be mined to recover change rationale cues and to link fixes to prior changes [35]. Previous IaC research has used Git history to empirically analyze how infrastructure configuration files and application source code change together over time in large-scale systems, revealing strong coupling and characteristic co-evolution patterns [37]. In software engineering more broadly, commit history has been leveraged to trace defect-inducing changes and study maintenance and evolution dynamics [35]. By analyzing commit-level context, researchers can infer not only when a change was made, but also plausible reasons for the change, which enables detection of smells that depend on configuration evolution or temporal patterns. In this thesis, we exploit commit history data to enable a context-aware detection of sustainability smells.

2.11 Code Smells

Code smells, a concept popularized by Martin Fowler in his seminal book "Refactoring: Improving the Design of Existing Code" [38], refer to any characteristic in the source code that suggests deeper issues, even though they do not prevent the program from functioning. These smells signal the need for refactoring to improve maintainability, readability, and quality. Code smells can arise in various forms, such as duplicated code, overly long methods, large classes, or complex inheritance hierarchies [39]. Beyond code, similar smells can manifest in configuration files, software models, or software model transformations where inefficient structures lead to poor maintainability and potential vulnerabilities [40, 41]. Smells in configuration files might include hardcoded values or redundant configurations, while smells in software models can undermine maintainability, introduce redundancy, or create inefficiencies. Smells can affect various aspects of software beyond just code quality, extending into areas like security and sustainability. When it comes to configuration files like IaC, security smells have been identified as potential vulnerabilities that could compromise system safety, such as hardcoded credentials or weak access controls [42]. These security smells serve as indicators of deeper structural problems that, if ignored, could lead to breaches or other severe security failures. In addition to security, energy and sustainability have emerged as critical considerations; some code smells can cause high energy consumption [43, 44].

2.12 LLM–based Code Smell Detection

Code smell detection has traditionally relied on heuristic and metric-based techniques, which encode expert knowledge into static rules over structural properties of source code, such as method length or complexity metrics [45]. Although these approaches are easy to interpret and integrate into static analysis tools, numerous studies report limited precision and recall, especially for smells whose manifestation depends on subtle semantic or contextual cues rather than simple metric thresholds [45, 46].

To overcome these limitations, recent work has explored machine learning and deep learning models that learn smell-indicative patterns directly from labeled code examples [46,47]. With the advent of pre-trained language models for code, researchers have begun to adapt large and small language models to smell detection using more parameter-efficient paradigms, reducing the need for task-specific architectures and full fine-tuning [45, 47].

PromptSmell leverages prompt learning over pre-trained models by converting multi-label smell detection into a multi-class problem and injecting code snippets into natural-language templates, enabling effective detection with relatively small labeled datasets [47]. EnseSmells instead combines deep representations from pre-trained code models with design-oriented metrics in an ensemble architecture, showing that fusing structural features and statistical semantics improves accuracy over prior deep-learning baselines [46]. Most recently, Zhang et al. conduct a comprehensive evaluation of parameter-efficient fine-tuning (PEFT) methods applied to both small and large language models for method-level smell detection, demonstrating that PEFT can match or surpass full fine-tuning while substantially reducing GPU memory consumption [45]. These findings suggest that applying LLM-based detection methods to IaC is both feasible and promising. In this thesis, we build on this emerging line of research by applying LLM reasoning to IaC scripts and commit histories, enabling detection of sustainability smells that static heuristics cannot reliably catch.

2.13 Gaps in Related Work

Despite extensive research on cloud performance, elasticity, workload-aware scheduling, IaC practices, and code smells, several gaps remain unaddressed. Prior studies on IaC practices and IaC smells [15,48–50] do not systematically examine sustainability issues encoded directly in IaC, and no existing work defines or evaluates code smells that affect sustainability in this context. Existing cloud sustainability guidelines are provider-specific and lack IaC-focused operationalization, while empirical IaC studies rarely incorporate commit-level context or workload-aware information when assessing configuration quality. Moreover, smell detection

methods in IaC largely rely on syntactic heuristics and have not been adapted to sustainability concerns. These gaps collectively motivate our investigation into defining, categorizing, and detecting sustainability smells in Terraform-based IaC.

CHAPTER 3 SUSTAINABILITY SMELLS

This chapter introduces the concept of sustainability smells in IaC and explains how a taxonomy of these smells was derived using a grounded theory approach supported by empirical analysis of real-world Terraform repositories. The goal of this work is to provide a comprehensive, data-driven understanding of sustainability issues in IaC. The chapter also describes the process of building a dataset of Terraform repositories used to anchor the taxonomy in actual infrastructure practice and ensure its relevance to real development workflows.

Figure 3.1 provides an overview of the methodology. In this study, we applied grounded theory techniques (open and axial coding) informed by vendor sustainability guidance, including the AWS Well-Architected Framework Sustainability Pillar and corresponding best-practice documentation from Azure and GCP [6–8]. We selected these sources because they provide concrete, widely adopted, practitioner-oriented recommendations that connect architectural decisions to resource efficiency and cost/energy considerations. We extracted candidate smells and defined corresponding code signatures to ensure they could be empirically recognized in codebases. To validate and contextualize our findings, we conducted a survey with industry practitioners and reported both the survey results and empirical findings from analyzing public repositories on GitHub.

3.1 Analyzing Sustainability Best Practices

To conduct a rigorous, well-structured qualitative analysis of cloud sustainability best practices, we employed Strauss and Corbin's version of Grounded Theory [51]. This approach was chosen due to its suitability for developing detailed categories from complex data sources and its emphasis on an iterative process between data collection and analysis. We began by selecting sustainability reports from AWS, Azure, and GCP as primary data sources, given their comprehensive guidelines on sustainable cloud practices.

The data sources for this study were selected based on their relevance to widely used cloud platforms like AWS, Azure, and GCP, which each publish sustainability guidance for cloud practices. These reports were obtained from each provider's official documentation and are structured around sustainable practices for cloud infrastructures, highlighting areas such as energy efficiency, resource management, and cost reduction. The AWS Well-Architected Framework's Sustainability Pillar outlines specific recommendations on regional resource selection, workload alignment, and energy-efficient architecture [6]. Similarly, Azure and GCP

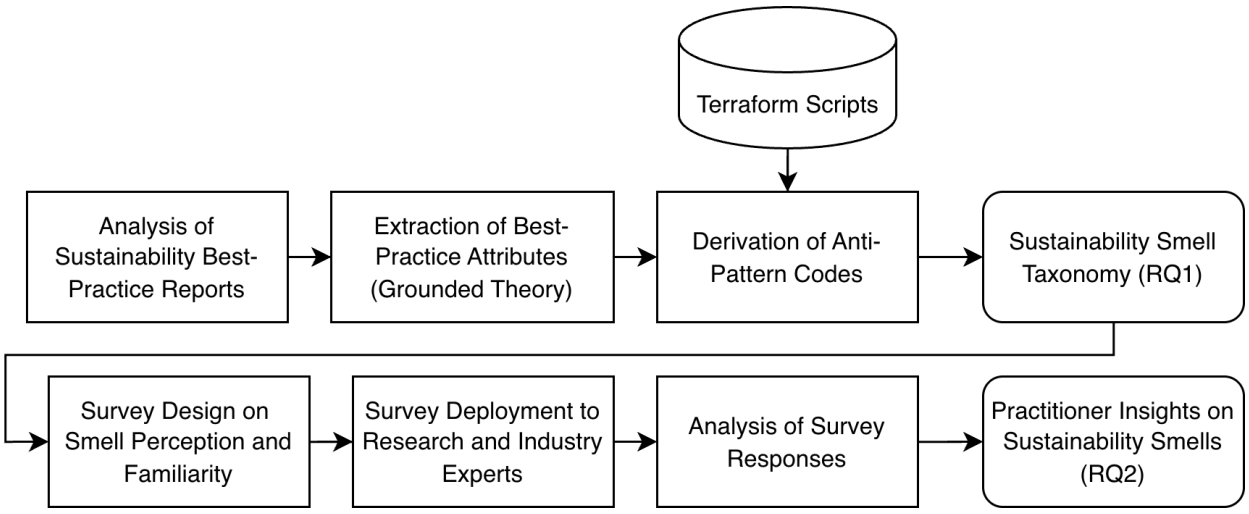

Figure 3.1 RQ1/RQ2 Methodology

sustainability best practices for Terraform emphasize optimizing cloud resources, workload scaling, and minimizing waste [7, 8].

Throughout the study, we adopted an iterative process that involved cycling between data collection and analysis, following the constant comparison and theoretical sampling techniques foundational to Strauss and Corbin's approach to Grounded Theory. Two researchers independently coded each document and then compared their codes to identify similarities and differences. Discrepancies were discussed and resolved through consensus meetings, allowing us to refine codes continuously. When new patterns or gaps emerged, we re-sampled from other sections of AWS, Azure, and GCP documents to validate findings, ensuring that emerging themes accurately captured the breadth of sustainable practices across platforms.

To ensure theoretical saturation, we conducted iterative rounds of open and axial coding and progressively expanded coverage to additional sections of the AWS, Azure, and GCP sustainability reports until no new categories, codes, or attributes emerged. In practice, after each coding cycle we sampled and coded further report sections and checked whether they introduced novel concepts beyond the existing codebook and attribute set. Saturation was evidenced by the recurrence of key attributes like **Runtime Dependency**, **Resource Context**, **Code Dependency**, and **Inherent Badness** across diverse best practices without additional variation in later cycles of analysis. This consistency indicated that our attributes sufficiently captured the variation in sustainability practices and that additional data was unlikely to yield new insights.

3.2 Prevalence Corpus (Dataset1) – Large-scale Terraform Repository Corpus

3.2.1 Dataset Overview and Provenance

The **Prevalence Corpus (Dataset1)** consists of publicly available, open-source Terraform repositories mined from GitHub. Repositories were selected based on the presence of Terraform configuration files (.tf) and excluded if they contained only templates, examples, or non-declarative artifacts. No restrictions were imposed on application domain, cloud provider, or organizational context, allowing the dataset to capture a broad spectrum of real-world IaC practices.

The resulting corpus comprises **28,327 Terraform scripts** drawn from **386 repositories** (after filtering) spanning diverse infrastructure domains, including compute provisioning, networking, storage, logging, and state management. Since the repositories are open source, they primarily reflect community-driven and industrial IaC usage rather than proprietary enterprise deployments.

The dataset covers multiple years of repository evolution as reflected in version control history. However, no assumptions are made about workload characteristics, deployment environments, or runtime behavior, as the analysis is strictly based on static IaC artifacts.

3.2.2 Building The Dataset

The **Prevalence Corpus (Dataset1)** comprises a total of **386 distinct GitHub repositories** containing Terraform files. This corpus was derived from an initial collection of **812** repositories selected based on their deployment configurations for the three major cloud providers: AWS, Azure, and GCP. The dataset collection was performed over different periods for each provider to ensure an up-to-date and comprehensive set of Terraform scripts. The breakdown of raw search results before filtering is as follows:

- AWS: 3,245 repositories.
- Azure: 1,308 repositories.
- GCP: 1,518 repositories.

The repositories were selected using the GitHub Code Search API, which allows for searching specific code snippets within the source files, rather than relying on metadata or file names. This method ensures that the dataset includes a wide variety of projects in which Terraform is used, regardless of the primary focus of the repository. We acknowledge that our dataset

primarily represents open-source Terraform scripts and may not fully capture proprietary industry practices. To ensure the quality and relevance of the dataset, we applied several filtering criteria:

- Size: The repository must have at least one file with non-zero size (i.e., > 0 KB).
- Originality: The repository must not be a fork to avoid duplicate or redundant data.
- Popularity: The repository must have at least 2 stars to indicate some level of community engagement.
- Data availability: The repository must be publicly accessible via the GitHub API, ensuring accessibility and replicability.
- Content: The repository must not be a course assignment, tutorial, or intentionally insecure project.

After applying these initial filters, the collected dataset comprises **401 AWS repositories** (after removing 12 that did not meet the content criterion), **137 Azure repositories** (after removing 6), and **274 GCP repositories** (after removing 7), totaling **812** repositories. We then applied additional repository-level filtering (e.g., duplicates, provider alignment, repository quality, and availability of usable commit history) to obtain the final **Prevalence Corpus (Dataset1)** of **386** repositories.

From these repositories, we extracted all Terraform files, obtaining a total of **28,327 Terraform scripts**. We sampled a subset of **1,860 Terraform scripts** from the dataset to facilitate a detailed qualitative analysis to identify code patterns that represent sustainability smells. For this purpose, we employed a stratified sampling approach to ensure diversity and representation across the collected repositories [52].

First, we defined each repository as a stratum, as each repository may have unique characteristics that could influence sustainability practices. Within each repository, we identified Terraform scripts as the primary unit of analysis. For each stratum (repository), we randomly selected **4–5 Terraform scripts** to ensure balanced representation while accounting for the variation in project complexity and configuration practices. The random selection was conducted using a random number generator to select scripts within each repository without bias. This approach allowed us to capture a diverse set of coding practices and sustainability smells while maintaining the integrity of each repository's specific configuration approach. By selecting scripts from different repositories within each stratum, we captured a

variety of coding practices and patterns, enhancing the robustness of our qualitative analysis. The result of this sampling process was a set of **1,860 Terraform scripts**. This subset was thoroughly examined to uncover specific code patterns and practices associated with sustainability smells.

3.3 Identifying Sustainability Smells

3.3.1 Conceptual foundation: Smells as indicators

Before deriving specific smells, we clarify our use of the term *sustainability smell* following established software engineering terminology. In Fowler's seminal work [38], a *code smell* is defined as "a surface indication that usually corresponds to a deeper problem in the system"—a symptom or warning sign that warrants inspection but is not necessarily problematic in all contexts. Code smells are *indicators* that suggest potential issues, not confirmed defects.

We adopt this paradigm for sustainability smells: a *sustainability smell* is an observable configuration pattern in IaC that serves as an indicator of potential resource inefficiency, cost waste, or environmental impact. Smells are not inherently "wrong" but represent configurations that deviate from established best practices and may lead to suboptimal outcomes depending on workload context, organizational constraints, and deployment architecture. This framing distinguishes sustainability smells from *antipatterns*, which represent confirmed bad solutions with empirically demonstrated negative consequences [13]. While antipatterns are retrospectively validated as harmful, smells are prospectively identified as warning signs requiring contextual evaluation.

3.3.2 Deriving smells from best practice violations

With a comprehensive list of best practices for sustainability derived from major cloud providers' recommendations, we can now proceed to identifying sustainability smells. The central premise is that each best practice extracted from AWS, Azure, or GCP has a corresponding *violation pattern* that signifies a failure to adhere to these guidelines. These violation patterns become candidate sustainability smells—configuration indicators that suggest a repository may not be following established sustainability guidance. Our objective is to leverage the dataset we have developed to uncover these violation patterns and compile a list of sustainability smells grounded in both provider recommendations and empirical evidence from real-world Terraform repositories.

By examining the key sustainability attributes we defined, we can identify specific violation

patterns that correspond to the failure to implement best practices effectively. For instance, a violation pattern associated with Runtime Dependency may involve neglecting dynamic scaling, leading to inefficient resource allocation—this violation pattern becomes a candidate sustainability smell. Once we identify these violation patterns, we delve into our dataset to uncover specific code signatures that embody these deficiencies. This involves qualitatively analyzing a subset of the dataset we collected, meticulously reviewing the structure, resource definitions, and configuration settings of each script over eight weeks to map out how these smell indicators manifest in actual code. The use of open and axial coding during this qualitative review provided a systematic way to extract recurring patterns and to group low-level codes into higher-level candidate smells [53, 54]. By establishing this connection, we create a comprehensive framework that links best practices, their corresponding violation patterns (smells), and the specific code signatures that reflect these potential inefficiencies, ultimately guiding our understanding of sustainability smells in cloud deployments.

In this stage, an important methodological consideration is why particular sustainability smells were selected while others were excluded. Our approach followed an open coding process, where each potential violation pattern was initially treated as a candidate smell indicator. As we examined both vendor best practices and actual Terraform code samples, we evaluated each candidate against two core criteria: (1) whether the configuration pattern could be reliably observed as a recurring indicator in our dataset, and (2) whether the pattern had a demonstrated relationship to increased resource consumption, cost inefficiencies, or preventable energy usage based on provider guidance or prior empirical work. The reliance on recurrence and demonstrable impact follows established qualitative and empirical practices for defining actionable code smells in IaC research [49,50]. Candidate smells that failed either criterion were discarded. For example, several practices related to organizational governance or business-level sustainability goals were excluded because they do not manifest directly in Terraform code and therefore cannot be consistently detected via repository analysis [15]. Similarly, some violation patterns appeared only once or were too context-specific to support generalization (i.e., they did not meet evidentiary saturation during coding) and were therefore rejected during coding [53, 54]. In contrast, smell indicators such as over-provisioning or the absence of autoscaling consistently appeared across multiple projects and were directly tied to measurable inefficiencies in prior literature, making them strong positive cases for inclusion; similar patterns have been reported in prior IaC-smell detection work and in studies linking provisioning decisions to energy/cost implications [49, 55, 56]. This positive–negative contrast informed the refinement of the final list of sustainability smells used in this study.

The criteria for choosing best practices comes from the established correlation between cloud infrastructure costs and energy consumption. Research by Horri et al. [55] highlights that

energy consumption in cloud environments can be modeled using cost-based models. Given this connection, our criteria for selecting best practices were based on the premise that non-alignment with a best practice would lead to increased costs and, by extension, higher energy consumption. To select best practices, we identified keywords frequently associated with cost inefficiencies, resource utilization, and energy consumption in cloud environments. These keywords include "overprovisioning", "underutilization", "inefficiency", "idle", "autoscaling", and "transfers". By focusing on these keywords, we were able to systematically identify practices that, when violated, would contribute to increased operational costs and energy consumption, thereby validating our criteria for selecting the best practices. The selected practices from AWS include SUS01-BP01, SUS02- BP01, SUS03-BP02, and SUS04-BP05.

3.4 Final Taxonomy of Sustainability Smells

Our analysis begins with open coding, going through all the best practices in the AWS sustainability pillar and identifying distinct phrases. During this phase, we systematically examined each best practice line-by-line, extracting only the parts that could reasonably manifest as observable configuration decisions in Terraform. These phrases were then translated into codes. For example, the phrase "scale your infrastructure dynamically to match supply of cloud resources" was coded as **Auto-Scaling**, highlighting a practice that encourages elasticity in resource allocation. Similarly, "Understand the devices and equipment used in your architecture and use strategies to reduce their usage" was coded as **SUS05-BP01 Use the minimum amount of hardware to meet your needs**, marking potential workload context dependencies that could influence cloud sustainability.

A key aspect of open coding was determining which best practices were suitable for translation into actionable codes and which were not. Some practices from AWS, GCP, and Azure explicitly pointed to infrastructure-level behaviors that could produce detectable patterns in Terraform (e.g., resource sizing, data transfer paths, storage configuration, lifecycle rules). These were included. Others, however, described high-level organizational, operational, or business-oriented recommendations such as adopting carbon-aware procurement strategies, negotiating sustainability focused SLAs, or establishing internal sustainability culture which cannot be mapped to concrete Terraform constructs. These were excluded because they either lacked code-level representations or would not produce consistent, machine-checkable signals.

To ensure comprehensive coverage, we also reviewed sustainability recommendations from other major cloud providers following the same criteria, such as GCP [7] and Azure [8]. Across providers, we consistently included practices that: (1) referenced infrastructure resources

directly, (2) could influence resource allocation or usage, (3) were expressed in a way that implied a "correct" and "incorrect" configuration pattern, and (4) appeared frequently enough across cloud guidance to signal general relevance. Practices that did not meet these criteria such as those tied purely to user behavior, billing strategy, or organizational planning were not converted to codes. See Table A.1 for examples of all the codes derived.

During open coding, we also captured positive and negative cases to refine the inclusion criteria. For example, "Use managed services" appears in several cloud guidelines but was excluded because it reflects architectural preference rather than a misconfiguration detectable in Terraform. In contrast, "right-size compute resources" consistently maps to explicit resource attributes (e.g., instance type, CPU count, storage size), making it suitable for coding. Likewise, "minimize data movement" produced clear signals related to region, zone, or networking configuration, whereas "optimize team processes to reduce waste" lacked any Terraform footprint, and was therefore set aside. These distinctions helped ensure that the extracted codes represent concrete, repeatable patterns linked to sustainability outcomes.

During open coding, we observed that sustainability best practices for cloud infrastructures can vary significantly. A best practice suitable for one type of infrastructure might be irrelevant or inefficient for another. For instance, in a known infrastructure setup, named Infrastructure **A**, where storage demands are stable and constant (e.g., always needing X storage), practices such as auto-scaling based on demand may be unnecessary. Conversely, for another Infrastructure **B**, where storage requirements fluctuate depending on workload, implementing demand-based resource scaling becomes essential to manage variability and avoid inefficiencies. This distinction highlights the importance of tailoring sustainability strategies to the specific needs and operational behaviors of each infrastructure type. Observing variability in sustainability best practices across cloud infrastructures highlighted the need to define specific attributes for these practices. These attributes help explain the contextual factors of an infrastructure that make a given best practice applicable. For example, Infrastructure **A** might have constant storage needs, whereas Infrastructure **B** experiences demand fluctuations, making certain practices like auto-scaling relevant only to **B**. This idea led to axial coding, where we defined attributes to capture these factors systematically, thereby identifying when and where sustainability practices are most effectively applied.

In the axial coding phase, we focused on identifying commonalities among initial codes generated during open coding to derive higher-level themes, or "attributes," that characterize sustainability best practices across cloud platforms. To start, we examined each code's relationship to broader sustainability goals, analyzing if it was influenced by runtime workloads, resource configurations, application code, or if it represented a universally inefficient practice.

For example, practices like “SUS02-BP01 Scale workload infrastructure dynamically” from AWS were linked with **Runtime Dependency** as they are designed to dynamically adjust resource allocation in response to workload fluctuations. Similarly, the codes of the best practices “CO:07 Optimize component costs” and “Choose the most suitable cloud regions" from Azure and GCP were assigned to a category indicating specific infrastructure configurations, leading to the **Workload Context** attribute. For another attribute we formed which is **Code Dependency**, we examined codes that are closely tied to the application's structure or software requirements. For instance, AWS's “SUS04-BP07 Minimize data movement across networks" hints at the need to minimize the total networking resources required to support data movement for the workload (which is controlled by the actual code of the running application on the cloud) pointing toward a dependency on the code structure rather than on infrastructure alone. Finally, we grouped universally inefficient practices, such as idle resources or excessive logging, under **Inherent Badness**, as these practices are considered bad for sustainability regardless of workload context.

After examining the dataset, we identified a set of sustainability smells that violate the best practices. To finalize the taxonomy, we applied the following selection criteria: (1) the smell must directly correspond to a recurring and detectable violation of at least one best practice; (2) the violation must be observable in Terraform code without requiring runtime monitoring; and (3) the issue must materially affect energy, resource usage, or cost. Codes that only represented high-level organizational strategies, business decisions, or operational processes without Terraform-level traces were excluded. For example, practices related to reviewing procurement policies or selecting carbon-aware regions were not converted into smells because they lack consistent code-level manifestations. Conversely, codes such as auto-scaling, resource sizing, data transfer patterns, and state configuration produced clear, repeated patterns in Terraform scripts that allow operationalization as sustainability smells.

During selection, we also evaluated borderline cases (positive and negative examples) to ensure that included smells were both technically measurable and conceptually meaningful. For instance, “Minimize Hardware Amount” appeared during coding but was not included as a smell because it does not map to a specific, consistent Terraform misconfiguration. Similarly, “Use managed services” was rejected because it reflects architectural choice rather than an identifiable anti-pattern in Terraform. In contrast, practices tied to instance sizing, logging configuration, lifecycle rules, and resource modularity consistently produced code patterns that could be checked across scripts, and therefore were retained as sustainability smells. These decisions ensured that the final taxonomy contains only smells that are both grounded in provider best practices and viable for automated or semi-automated detection.

We refer to sustainability smells from now on with the notation (SSx) where x is the number of the smell. The identified sustainability smells include the following:

Over-Provisioning Resources (SS1):

Over-provisioning occurs when Terraform scripts allocate compute, memory, or storage resources that significantly exceed the actual needs of the workload. This smell is frequently introduced during early development and testing, where developers intentionally choose large instance types to avoid performance issues, and later forget to adjust them when workloads stabilize [57]. It can also appear in legacy configurations that were sized for older, heavier workloads but never revisited after optimizations or changes in usage patterns [58]. In multi-team environments, over-provisioning may result from conservative assumptions or limited visibility into real workload behavior [59].

The negative effects of over-provisioning are twofold. First, it leads to unnecessary financial cost because larger resources typically have a much higher hourly rate [57]. Second, it increases environmental impact since unused CPU cycles or memory still consume power even when idle; studies and surveys of cloud energy consumption emphasize that excess provisioning increases overall data-center and cloud energy use [60].

For example, the snippet below shows a virtual machine with an excessively large instance type:

Listing 3.1 Example of SS1: over-provisioned compute instance

```
resource "azurerm_virtual_machine" "inefficient_vm" {
    name                = "example-vm"
    location            = "East US"
    vm_size             = "Standard_D32s_v3" # Overprovisioned
}
```

Right-sizing—regularly reviewing and adjusting instance types—is therefore essential to aligning infrastructure with actual workload demands [57].

Lack of Auto-Scaling (SS2):

This smell occurs when Terraform configurations rely on fixed or manually specified resource counts rather than using autoscaling mechanisms provided by cloud providers. It commonly arises in systems that were initially designed with predictable workloads but later experience growth or variability, or in cases where developers assume that a fixed number of instances

is “good enough” without evaluating real usage patterns [61]. It may also occur when teams are unfamiliar with autoscaling resources or consider them complex to configure.

Without auto-scaling, infrastructure cannot adapt dynamically to changes in demand. During low-traffic periods, fixed provisioning results in underutilized resources that continue to consume energy despite minimal workload. Conversely, during peak demand, the same fixed configuration may become insufficient, forcing operators to over-provision instances preemptively just to avoid performance issues [61]. Both scenarios introduce inefficiencies: either wasted compute cycles or unnecessary large baselines that increase cost and resource consumption.

Listing 3.2 Example of SS2: fixed resource count (no autoscaling)

```
resource "aws_instance" "app" {
    count = 5 # Fixed number of instances
    ami = "ami-0c55b159cbfafe1f0"
    instance_type = "m5.2xlarge"
}
```

While autoscaling may be intentionally avoided in certain regulated or latency-sensitive environments, SS2 focuses on cases where autoscaling is applicable but omitted, resulting in avoidable inefficiencies [8].

Ignoring Resource Lifecycles (SS3):

This smell appears when Terraform resources lack explicit lifecycle rules that govern how they are created, replaced, or destroyed. It typically occurs in early-stage deployments where lifecycle behavior is not yet a concern, or in teams with limited familiarity with Terraform’s lifecycle features [62]. It also commonly appears when infrastructure changes evolve incrementally over time, but lifecycle rules are never revisited to reflect those changes.

Without lifecycle rules, Terraform may behave inefficiently whenever a configuration update is applied. For example, resources can be unintentionally replaced instead of updated in place, leading to unnecessary recreation. This increases deployment times, may temporarily allocate duplicate resources, and in some cases triggers unintended downtime [63]. Retaining unused or outdated resources due to missing lifecycle constraints can also increase long-term cost and energy usage.

Properly defined lifecycle rules help ensure predictable behavior, avoid redundant provisioning, and reduce operational overhead [62].

Listing 3.3 Example of SS3: missing lifecycle safeguards

```
resource "azurerm_managed_disk" "example" {
    name                 = "example-disk"
    storage_account_type = "Standard_LRS"
    create_option        = "Empty"

    lifecycle {
        create_before_destroy = false
    }
}
```

Excessive Logging (SS4):

This smell occurs when systems generate or retain logs at a level of detail that exceeds operational requirements. Developers often enable verbose logging during debugging or testing phases, and these settings unintentionally persist into production [64]. As workloads grow, the volume of logged events increases significantly, making excessive retention periods particularly costly. This is especially common in high-throughput environments where log generation is continuous.

The negative impact of excessive logging arises from both storage and processing. Storing high-volume logs for long periods increases storage consumption, while maintaining or querying these logs requires additional compute resources. Cloud provider documentation and guidance explicitly call out log ingestion, storage, and retention as drivers of monitoring costs [65, 66].

Listing 3.4 Example of SS4: excessive log retention

```
resource "aws_cloudwatch_log_group" "detailed_logs" {
    name = "detailed-log-group"
    retention_in_days = 365 # Long period for detailed logs
    ...
}
```

Limiting retention to the minimum required for compliance or operational needs can significantly reduce resource consumption [64].

Unoptimized Data Transfers (SS5):

This smell appears when data frequently flows between resources located in different regions or zones without a clear architectural justification. It often arises as systems grow organically, e.g., when developers deploy new components in a separate region for testing, or when replication and backup mechanisms are introduced without revisiting overall placement decisions [60]. Multi-region deployments can also unintentionally produce this smell if data paths are not carefully evaluated.

Cross-region or cross-zone transfers increase latency, incur additional network egress costs, and use more energy compared to local data movement. Energy and sustainability literature for data centers highlights that unnecessary data movement increases the system's energy footprint because long-distance transfers require additional networking infrastructure and multiple intermediate hops [60, 67].

Listing 3.5 Example of SS5: potential cross-region or cross-zone data transfers

```
resource "google_compute_instance" "example" {
    name         = "example-instance"
    machine_type = "n1-standard-1"
    zone         = "us-west1-a"
    # Data frequently transferred to another region
}
```

Co-locating compute and storage resources and revisiting replication or backup strategies can significantly reduce this overhead [6].

Poor State Management (SS6):

This smell occurs when Terraform state is stored locally, inconsistently configured, or not backed by a reliable remote backend. It is common in early prototypes, small projects, or teams new to Terraform, where local state seems sufficient initially [68]. As the infrastructure grows, however, inconsistent state handling can introduce drift, corrupt state files, or cause resources to be recreated unintentionally.

Poor state management leads to inefficiencies when Terraform cannot accurately determine the current state of the infrastructure. This may result in repeated creation or modification of resources that have not actually changed. Local state files can also be accidentally deleted or overwritten, causing Terraform to interpret the infrastructure as missing and recreate

resources unnecessarily. Even with a remote backend, missing locking mechanisms or inconsistent key prefixes may result in parallel updates or state fragmentation [69].

Listing 3.6 Example of SS6: missing remote backend (local state by default)

```
# No remote backend is configured; Terraform uses local state by default.
# (In practice, this is the default behavior when no backend block is present.)
```

Proper remote backends, state locking, and clear naming conventions ensure consistency and reduce the operational overhead associated with managing infrastructure state [68, 69].

Non-Modular Configurations (SS7):

This smell appears when Terraform configurations are written in large, monolithic files without logical separation into modules. It typically emerges in rapid prototyping environments, small teams, or early-stage projects where infrastructure grows quickly and organically [70]. As more resources are added, developers may continue adding them to a single file for convenience, eventually resulting in a configuration that becomes difficult to navigate or maintain.

Non-modular configurations negatively affect maintainability and long-term sustainability. Without modules, it becomes harder to reuse common patterns, enforce best practices, or isolate components for separate management. This often leads to duplicated definitions across different environments and makes updates more error-prone. Large monolithic configurations also slow down review processes and discourage refactoring, increasing the likelihood of persistent inefficiencies as infrastructure evolves [70, 71]. Adding to this, Terraform plan and apply commands will take longer because they have to process the entire infrastructure graph for every change, even small ones.

Listing 3.7 Example of SS7: monolithic configuration (no modularization)

```
resource "google_compute_network" "main" { ... }
resource "google_compute_subnetwork" "example" { ... }
# Many more resources follow...
```

Modularizing through Terraform modules helps establish clearer boundaries, improves readability, and simplifies future modifications, contributing to a more sustainable long-term infrastructure design [7, 70].

3.5 Validation of Sustainability Smells

To validate our identified sustainability smells, we engaged two independent raters who were not involved in this study. These raters, possessing expertise in IaC, were tasked with assessing whether the identified smells corresponded to the associated best practices outlined in industry reports. For this purpose, we supplied the raters with the names of the smells, an example of each, and the corresponding best practice. Each rater independently evaluated the alignment of the smells with the provided best practices. Both raters consistently associated each of the seven sustainability smells with the same best practice. The evaluation yielded a Cohen's Kappa score of 1.0, reflecting perfect agreement between the raters.

To understand how practitioners perceive the impact of these sustainability smells, we conducted a survey targeting practitioners with experience in Terraform-based IaC. We recruited participants with IaC experience working with major cloud providers (AWS, Azure, and GCP) and included researchers in sustainability and cloud computing to broaden perspectives, for a total of 19 participants. The survey comprised 22 questions (Google Forms) combining multiple-choice and open-ended items; it began with demographic and experience questions and then presented code snippets illustrating each sustainability smell. For each smell, participants reported how frequently they encounter the pattern in practice and whether they perceive it as a bad practice (i.e., evidence of face validity). We analyzed the closed-ended responses quantitatively and used open-ended responses to refine definitions and examples.

Most of the survey participants have substantial experience in programming, with 5 to 10 years in general development and an equal range of experience specifically in Infrastructure as Code (IaC). Figure 3.2 shows the IaC years of experience (YOE) for the survey participants.

We found that **SS1** and **SS7** are the most frequently encountered smells, with 52.6% of the participants often or always facing these issues. **SS3** and **SS2** are also common, with 36.8% of participants reporting that they encounter them sometimes. Around 26.3% of the participants often encounter **SS2**, while 21.1% often encounter **SS3**. **SS4** and **SS5** are less frequently encountered but still significant, with more than a third of the participants encountering these sometimes or often. **SS6** practices are another notable issue, with 31.6% of participants facing it often and 26.3% always facing it. Our replication package contains all the detailed responses for all the identified smells.

Figure 3.3 shows the participants' perception of sustainability smells being actually bad for sustainability, it is an answer to the question "Do you perceive this pattern as a bad practice?". The data indicates that several identified sustainability smells in Terraform re-

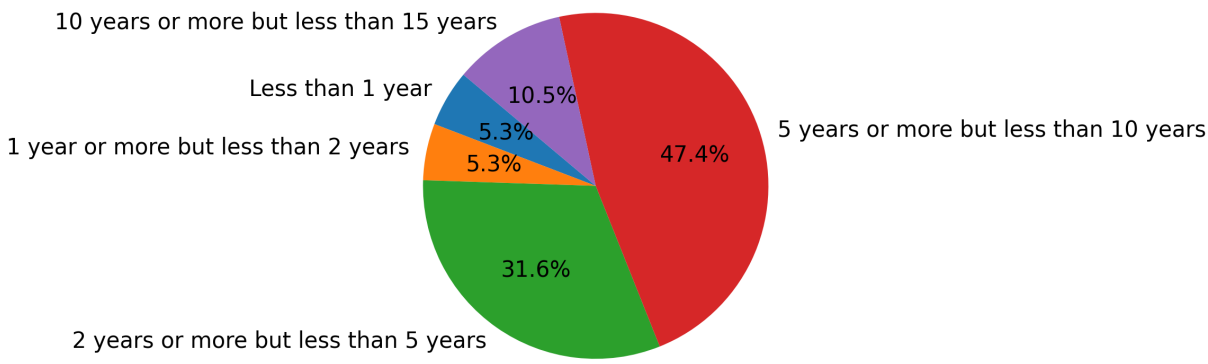

Figure 3.2 Survey participants YOE using IaC tools

ceived a majority vote, signaling consensus that they are generally considered bad practices. Specifically, the following percentages of participants agreed that each sustainability smell represents a negative practice: **SS1** (94.7%), **SS2** (78.9%), **SS3** (47.4%), **SS4** (68.4%), **SS5** (57.9%), **SS6** (47.4%), and **SS7** (78.9%). These results highlight the prevalence of certain practices viewed as unfavorable, particularly for **SS1**, **SS2**, and **SS7**, which received notably high levels of agreement.

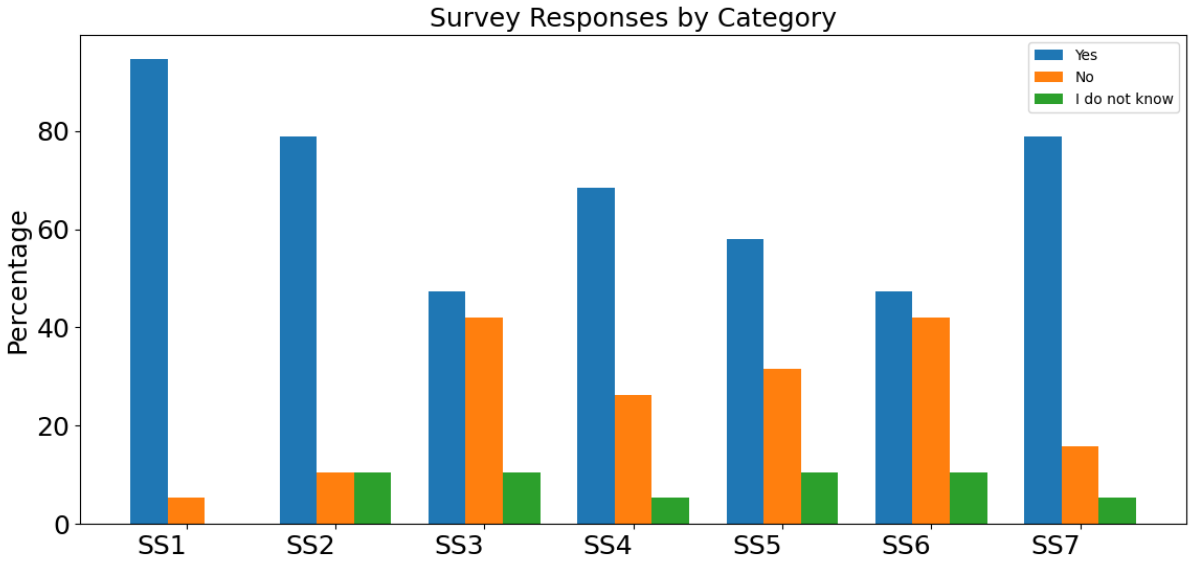

Figure 3.3 Survey Participants Perception of Sustainability Smells

3.6 Sustainability Smells Categorization

We decided to categorize the sustainability smells we derived to better understand their different impacts and contexts, enabling more targeted and effective mitigation strategies for enhancing cloud deployment sustainability. To categorize sustainability smells credibly and systematically, we adopted a rigorous research methodology. We use hierarchical clustering because it is a powerful statistical method used to group similar data points based on their characteristics, and it works by recursively merging or splitting clusters based on their similarity, forming a tree-like structure known as a dendrogram [72]. We begin by reviewing the attributes of the sustainability smells we previously defined, analyzing each one to identify the specific attributes that characterize it. The results of this analysis are presented in Table II, where each row corresponds to a sustainability smell. Each cell within the table indicates whether a particular attribute is present for that smell, with a value of 1 indicating the presence of the attribute and a value of 0 indicating its absence. Then we calculate the semantic similarity between all the combinations of sustainability smells pairs. The similarity is in a form of a matrix that was populated by comparing the attributes of each smell. Based on our definition of similarity, if two smells have the same attributes, their similarity score would be 1. We used only 0 and 1, where 0 indicates no similarity and 1 indicates complete similarity. It is only 0 or 1 because the similarity between two smells is not some quantity that can be numerically quantified as it is based on a complete match of attributes between two smells. This allows for the identification of natural groupings within the data, providing insights into the relationships and patterns among different sustainability smells. This technique is particularly effective for organizing complex datasets, such as sustainability smells in IaC scripts, where there are numerous dimensions and attributes to consider.

For each sustainability smell, we use the defined set of sustainability best practices attributes to help describe its characteristics. To ensure clarity and transparency, we organized the attributes and their corresponding values (1 if the attribute exist in the smell and 0 otherwise) for each sustainability smell in a table. An example is shown in Table 3.1. We create a similarity matrix to quantify the similarity between each pair of sustainability smells. The matrix is shown below, each row and column in the matrix represents a specific sustainability smell. The entries (0 or 1) indicate whether a pair of smells shares significant similarities (1) or not (0). The similarity matrix between each pair of sustainability smells is shown below:

Table 3.1 Attributes of Sustainability Smells

Smell (SS)	Runtime Dependency	Resource Context	Code Dependency	Inherent Badness
SS1	1	1	0	0
SS2	1	1	0	0
SS3	0	0	0	1
SS4	0	0	0	1
SS5	0	0	1	0
SS6	0	0	0	1
SS7	0	0	0	1

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{3.1}$$

We use agglomerative hierarchical clustering, a bottom-up approach where each data point starts in its own cluster, and pairs of clusters are merged as we move up the hierarchy [73]. This method is particularly suited for our needs as it does not require us to predefine the number of clusters. We constructed a dendrogram to visualize the clustering process. The dendrogram illustrates the nested grouping of sustainability smells based on their similarities. At the bottom of the dendrogram, each leaf represents an individual sustainability smell. As we move up, nodes represent clusters formed by merging pairs of smells or clusters. Based on the dendrogram we can interpret the clustering results as follows:

- **Category 1: General Sustainability Smells:** Can be considered the rule of thumb (SS3, SS4, SS6, SS7).
- **Category 2: Demand Sustainability Smells:** Requires data about the actual demand of resources for the workload whether this comes on runtime or based on predefined requirements so that the recommendations can make sense (SS1, SS2).
- **Category 3: Application Sustainability Smells:** Requires data about the actual code the infrastructure will run (SS5).

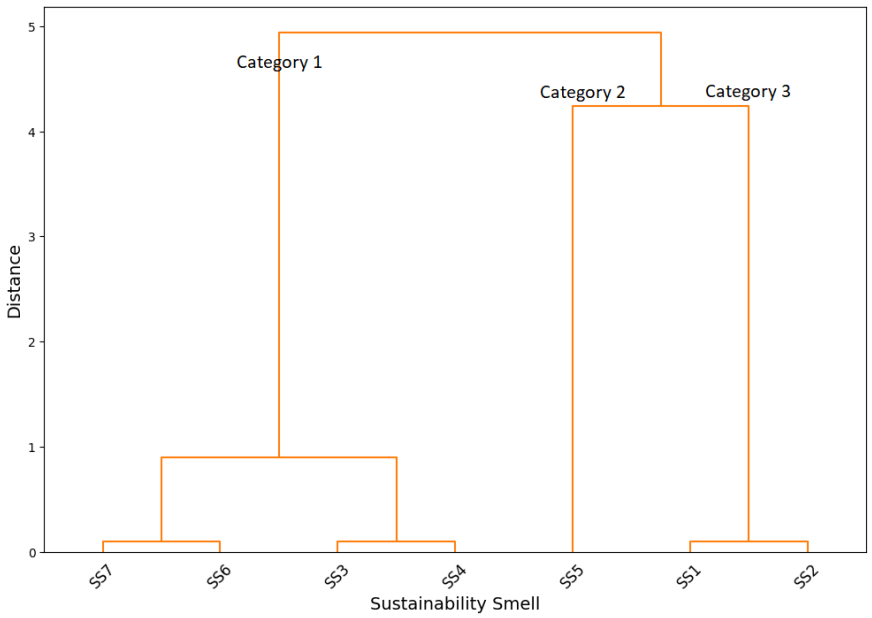

Figure 3.4 Hierarchical Clustering Dendrogram for Sustainability Smells

Summary of Research Questions

RQ1: What sustainability smells occur in IaC scripts?

Through a grounded analysis of sustainability guidance from AWS, Azure, and GCP, combined with iterative coding of 1,860 Terraform scripts from 395 repositories, we derived a taxonomy of sustainability smells grounded in both cloud best practices and real-world IaC usage. The resulting smells capture recurrent code-level patterns that misalign with principles of resource efficiency, right-sizing, and waste minimization. Each smell is defined through a clear conceptual description, an associated code signature, and supporting examples validated across multiple providers. This process yields a concise, empirically grounded set of sustainability smell categories.

RQ2: How do practitioners perceive the proposed sustainability smells?

A survey of 19 experienced IaC practitioners demonstrates strong support for the relevance, clarity, and practical importance of the proposed smells. Participants validated the conceptual soundness of the taxonomy, rated most smells as moderate to highly impactful on sustainability, and provided qualitative feedback that led to refinements in naming, scope, and definition boundaries. Their assessments confirm that the smells reflect meaningful sustainability issues observed in real-world IaC development and operations, thereby establishing face and construct validity for the taxonomy.

CHAPTER 4 MEASURING THE PREVALENCE OF SUSTAINABILITY SMELLS

The goal of this chapter is to measure how frequently the sustainability smells introduced in Chapter 3 occur in real-world Terraform repositories. To do so, we operationalize each smell definition into an observational labeling step that maps each Terraform file f to a binary vector $\mathbf{y}$ indicating whether the file exhibits one or more smell-indicative configuration patterns. This labeling is used exclusively for prevalence measurement (via aggregation across files and repositories), not for design-time guidance. The challenge arises because IaC smell manifestations are highly contextual: identical syntactic structures may represent sustainable or unsustainable choices depending on workload requirements, deployment architecture, or organizational constraints. As a result, purely syntactic rules struggle to infer developer intent, resource semantics, or the implicit assumptions encoded in infrastructure design.

The core objective of this chapter is not to determine whether an infrastructure deployment is sustainable, nor to enable proactive prevention or design-time warnings. Instead, the goal is to measure the prevalence of sustainability smells in real-world Terraform repositories as accurately as possible, given the inherent limitations of static IaC analysis. Sustainability smells are treated as post-hoc indicators observed in existing infrastructure configurations, and version control history is leveraged solely to provide additional context that helps disambiguate whether an observed pattern plausibly reflects an inefficiency. The methods presented in this chapter are therefore observational in nature and are intended to support empirical analysis of how frequently sustainability smells occur in practice, rather than to guide design-time decision-making or operational optimization.

In this chapter, we assume that Terraform scripts contain sufficient information to reveal sustainability-relevant decisions, but we acknowledge that intent is often distributed across modules, variables, and commit history. We define *context* as any information beyond the literal code snippet including variable files, module abstractions, architectural constraints, and commit-level rationale that influences whether a configuration represents a sustainability smell.

4.1 Evaluation Benchmark (Dataset2) – Labeled Ground-Truth Dataset

The Evaluation Benchmark (Dataset2) is a labeled dataset of 148 Terraform repositories derived from prior work [74]. It provides repository-level ground truth for four sustainability

smells (SS1, SS3, SS4, SS5) and is used solely for accuracy evaluation and method verification. This benchmark is **not** used for ecosystem-wide prevalence estimation, which relies instead on the unlabeled Prevalence Corpus (Dataset1) described in Section 4.

To support retrospective measurement of sustainability smell prevalence, we leverage an existing, publicly available dataset introduced by Bolhuis et al. [74], which investigates cost management practices in Terraform-based IaC repositories. That work presents a catalog of recurring *cost patterns* and *cost anti-patterns* identified through thematic analysis of cost-aware commits in real-world GitHub repositories. The dataset reports repository-level occurrences of these anti-patterns, based on empirical analysis of Terraform configuration changes and associated version control history.

4.1.1 Conceptual Distinction: Smells vs. Antipatterns

Following established software engineering literature [13, 38], we distinguish between *code smells* and *antipatterns* as follows:

- **Code smell**: A surface indication or symptom suggesting a potential design problem or a "warning sign" that warrants further inspection but is not necessarily problematic [38]. Smells are *indicators* of possible issues.
- **Antipattern**: A commonly used solution or practice that appears beneficial initially but ultimately produces negative consequences [13]. Antipatterns are *recurring bad solutions*, not symptoms.

The sustainability smells defined in Chapter 3 follow the code smell paradigm: they are *observable configuration patterns that serve as indicators of potential resource inefficiency.* In contrast, the cost anti-patterns from Bolhuis et al. [74] represent *empirically validated bad practices* associated with increased cloud cost. These are fundamentally different concepts.

4.1.2 Justification for Using Antipatterns as Smell Evidence

Despite this conceptual distinction, cost antipatterns can serve as empirical evidence for sustainability smells under specific conditions. The key insight is that antipatterns represent *instances where the symptom (smell) has been confirmed to lead to negative outcomes.* When a repository exhibits a cost antipattern such as "expensive instance" or "over-provisioned resources," this provides retrospective evidence that:

1. The underlying configuration pattern (smell) was present

2. The pattern materialized into an observable inefficiency (cost increase)

3. Developers recognized it as problematic (evidenced by subsequent mitigation attempts)

This relationship is analogous to using confirmed disease cases as ground truth for studying diagnostic symptoms: the presence of a diagnosed condition (antipattern) validates that the symptoms (smells) were indeed indicative of a problem. However, this validation is *retrospective and partial* because not all smells lead to confirmed antipatterns, and not all antipatterns correspond to observable smell indicators in static IaC artifacts.

Patterns capture recurring, proactive solutions adopted by developers to manage or reduce cloud cost, whereas anti-patterns capture recurring, empirically observed practices that are associated with increased cost. In this thesis, only anti-patterns are used as external evidence for repository-level ground-truth anchoring; patterns are shown here solely to clarify the conceptual distinction between preventive practices and problematic configurations.

To concretely distinguish between patterns and anti-patterns as defined by Bolhuis et al., Listings 4.1 and 4.2 present representative Terraform excerpts adapted from the original catalog. These examples are included for conceptual clarification only.

Listing 4.1 Example of a cost pattern: lifecycle rules to reduce storage cost

```
resource "aws_s3_bucket_lifecycle_configuration" "example" {
  bucket = aws_s3_bucket.bucket.id

  rule {
    id     = "log"
    status = "Enabled"

    expiration {
      days = 90
    }

    transition {
      days          = 60
      storage_class = "GLACIER"
    }
  }
}
```

Cost anti-patterns appear exclusively in the **Evaluation Benchmark (Dataset2)** because they originate from prior work [74] and are used only to anchor repository-level labels for accuracy evaluation. In contrast, the **Prevalence Corpus (Dataset1)** contains no patterns or

Listing 4.2 Example of a cost antipattern: overprovisioned compute instance

```
resource "google_compute_instance" "example" {
  name         = "example"
  machine_type = "a3-highgpu-8g"
}
```

anti-patterns and does not rely on labeled data; it is used exclusively for large-scale retrospective prevalence measurement across all seven sustainability smells in Terraform repositories.

4.1.3 Mapping Methodology and Theoretical Basis

Anti-patterns are **not assumed to directly correspond** to sustainability smells. Instead, we establish a **manual, conservative, and explicitly partial mapping** between a subset of cost anti-patterns and four sustainability smells (SS1, SS3, SS4, SS5), grounded in the following theoretical rationale:

1. **Symptom-to-outcome correspondence**: When Bolhuis et al. identify an "expensive instance" antipattern through empirical analysis of commit history, this indicates that developers observed and addressed resource over-provisioning. The antipattern's presence confirms that the underlying configuration pattern (SS1: over-provisioning smell) manifested with sufficient severity to warrant corrective action.

2. **Static observability**: Both sustainability smells and cost antipatterns are identifiable through static IaC analysis without requiring runtime measurements. This shared observability enables transitive inference: if a repository exhibits static evidence of a cost antipattern, it also exhibits the configuration pattern that constitutes the smell.

3. **Conceptual overlap with qualifications**: The mapping is restricted to cases where the antipattern definition semantically overlaps with the smell definition. For example, "Over-provisioned Resources" (antipattern) directly aligns with "Over-Provisioning Resources" (SS1 smell), as both describe excessive capacity allocation. However, we explicitly document cases of *partial* or *approximate* overlap (see Table 4.1), acknowledging that not all dimensions of a smell may be captured by the corresponding antipattern.

This mapping is used solely for repository-level ground-truth inheritance and does not imply: (a) equivalence in scope or severity, (b) completeness of smell manifestations, (c) identical labeling criteria, or (d) correspondence in frequency of occurrence within a repository. The

inherited labels indicate only that *at some point in the repository's evolution, a configuration pattern corresponding to the mapped smell was present and empirically associated with cost inefficiency.*

Cost anti-patterns are used in this work because they represent empirically observed classes of inefficient infrastructure configurations that can be catalogued at scale through static IaC analysis. While cost is not equivalent to environmental impact, these anti-patterns provide a consistent and reproducible reference point for prevalence measurement. Accordingly, antipattern-derived labels are not interpreted as evidence of actual cost overruns, energy waste, or runtime inefficiency, but only as indicators that a repository has exhibited at least one configuration belonging to a known class of cost-inefficient IaC practices.

In this work, we focus exclusively on *cost anti-patterns*, as opposed to cost patterns, since anti-patterns represent potentially harmful or inefficient configurations. From the full catalog, we manually identified a subset of anti-patterns that could be meaningfully mapped to the sustainability smells defined in Chapter 2, using the pattern-to-smell mappings shown in Table 4.1. This mapping process was performed manually, and only four sustainability smells could be reliably matched to the available cost anti-patterns. As a result, the **Evaluation Benchmark (Dataset2)** is intentionally limited to only four smells (SS1, SS3, SS4, SS5).

4.1.4 Constructing the Evaluation Benchmark (Dataset2)

We filtered the catalog to retain only Terraform-related instances, discarding examples associated with other IaC tools. For each retained entry, we extracted the commit URLs provided by the original dataset and identified the corresponding repositories. Using the established pattern-to-smell mappings, we constructed a dataset that associates each repository with the set of sustainability smells implied by the mapped anti-patterns. This process resulted in the **Evaluation Benchmark (Dataset2)**: a curated dataset of 148 Terraform repositories with explicit ground-truth labels.

After completing the mapping process, we observe that only four smells (SS1, SS3, SS4, and SS5) are represented in the resulting benchmark. While this limits coverage across all seven defined smells, it does not invalidate accuracy evaluation for the covered smells, as this benchmark is currently the only publicly available resource that provides empirically verified ground-truth labels relevant to sustainability smells in IaC. Consequently, the **Evaluation Benchmark (Dataset2)** serves as a valuable resource for assessing prevalence-estimation error for these four smells, while the **Prevalence Corpus (Dataset1)** is used separately to measure prevalence across all seven smells without ground-truth validation.

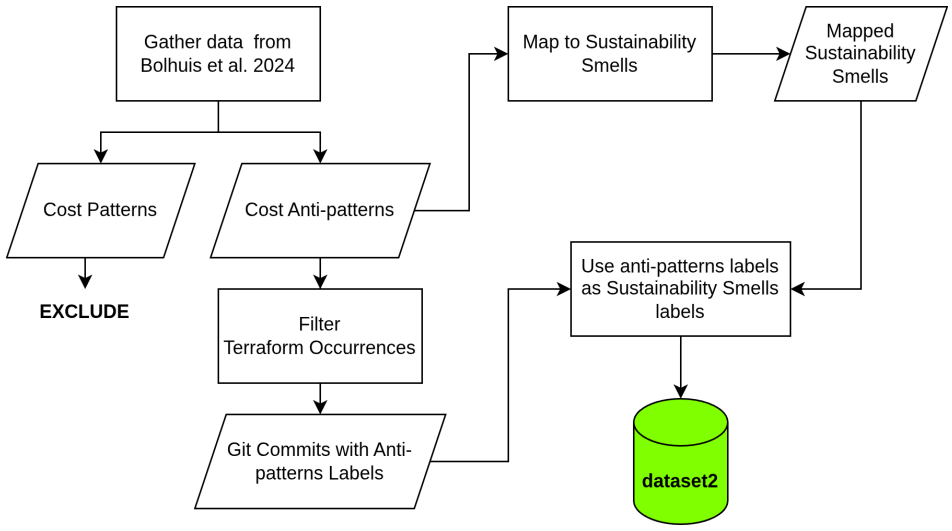

Figure 4.1 Evaluation Benchmark Construction Process

Ground truth labels in the **Evaluation Benchmark (Dataset2)** are assigned at the *repository level*. By transitive reasoning, if a repository is reported by the original dataset [74] as containing a specific cost anti-pattern, and that anti-pattern is mapped to a sustainability smell, then the repository is labeled as containing that smell. No additional inference is made regarding the severity, frequency, or runtime impact of the smell within the repository. This labeling strategy directly inherits the empirical findings of the original dataset and enables comparative accuracy evaluation without re-identifying anti-patterns from scratch.

Importantly, Git commit history is not used to establish ground truth labels in the **Evaluation Benchmark (Dataset2)**. Commit-level information provided by the original dataset serves solely as supporting evidence that a given anti-pattern occurred in the repository at some point in its evolution. In subsequent analyses, commit messages and diffs are used only to provide historical context when interpreting prevalence measurements and to reduce misclassification that may arise from purely syntactic inspection. Commit data is therefore incorporated for retrospective contextualization, not for labeling, prediction, or assessment of sustainability outcomes.

Refer to Figure 4.1 for a detailed process of constructing the Evaluation Benchmark (Dataset2).

4.2 Datasets and Their Functions

Although the Prevalence Corpus (Dataset1) and Evaluation Benchmark (Dataset2) are introduced in separate sections, they serve fundamentally different and non-overlapping roles in the study. The **Prevalence Corpus (Dataset1)** (**386 repositories** containing **28,327**

Table 4.1 Mapping of Antipatterns to Sustainability Smells

(Anti)Pattern	**SS Match**	**Match Type**	**Reasoning**
Budget	None	None	No direct match. (Anti)pattern definition: "Use budgets to receive alerts about charged and forecasted costs and control spending."
Spot instances	SS1	Part of SS	Spot instances can reduce over-provisioning for interruptible workloads.
Object storage lifecycle rules	SS3	Direct	Similar definitions.
Expensive instance	SS1	Part of SS	Expensive instances as part of over-provisioned resources.
Old generation	None	None	No direct match. (Anti)pattern definition: "Using newer resource generations gives similar performance for lower cost."
Expensive storage type	SS1	Part of SS	Expensive storage as part of over-provisioned resources.
Expensive network resource	SS1	Part of SS	Expensive network resources (e.g., NAT gateways, subnets, elastic IP addresses) as part of over-provisioned resources.
	SS5	Part of SS	Expensive network resources can increase inter-region/unnecessary data transfers. However, the (anti)pattern data mostly focuses on network resource creation/management rather than access patterns.
Over-provisioned Resources	SS1	Direct	Similar definitions.
Expensive DynamoDB	SS1	Part of SS	Expensive DynamoDB tables as part of over-provisioned resources.
Expensive monitoring	SS4	Approximate	The (anti)pattern focuses on both expensive monitoring solutions and logs while the SS only focuses on logs.
Cost report	None	None	No direct match. (Anti)pattern definition: "Cost report elements can be used to obtain information on the actual spendings over a period of time."

Interpretation: This table maps cost antipatterns (confirmed bad solutions) to sustainability smells (configuration indicators) based on conceptual overlap. **Match Types** indicate the strength of alignment: *Direct* denotes near-identical definitions; *Part of SS* indicates the antipattern captures one manifestation of a broader smell; *Approximate* signals partial overlap with definitional gaps. If a repository in Bolhuis et al. exhibits an antipattern, we transitively infer the presence of the mapped smell, interpreting the antipattern as retrospective evidence that the configuration pattern (smell) manifested with observable consequences. This inference is conservative and does not imply that all instances of the smell correspond to antipatterns, nor that smell severity equals antipattern impact.

Terraform scripts) is unlabeled and used to measure the prevalence of all seven sustainability smells across a broad set of Terraform repositories. The **Evaluation Benchmark (Dataset2)** (**148 repositories**), derived from prior work [74], provides repository-level ground-truth labels for a limited subset of four smells and is used exclusively for accuracy evaluation and method verification. The **Evaluation Benchmark (Dataset2)** is not used for prevalence measurement, and the Prevalence Corpus (Dataset1) is not used for performance assessment. Table 4.2 summarizes the roles of both datasets.

Table 4.2 Summary of datasets and their distinct roles

Dataset	**Size**	**Labeling**	**Function**
(1) Prevalence Corpus	386 repos	Unlabeled	Ecosystem-wide prevalence measurement
(2) Evaluation Benchmark	148 repos	Labeled	Performance and method evaluation

4.3 Regex-based Method

To measure the prevalence of sustainability smells, we developed tailored regular expression (regex) patterns based on specific attributes of each smell. Regex serves as the natural baseline for this chapter because it represents the simplest, most interpretable, and most widely used rule-based baseline in IaC smell research. It requires no training data, scales cheaply, and reflects the established practice in prior work that operationalizes smells through static heuristics. Evaluating more advanced methods against a regex baseline allows us to quantify the value of context-awareness and reasoning while maintaining comparability with earlier studies.

First, we reviewed each sustainability smell and determined keywords, syntax structures, and configuration patterns that could serve as reliable indicators. Each regex pattern was iteratively refined by testing it against a subset of scripts from our dataset. After initial pattern development, we conducted a manual validation step to assess labeling quality for prevalence measurement. A random sample of scripts matched by each pattern was manually reviewed to confirm that the matched smell indicators aligned with the defined criteria. We further summarized this validation using a precision-style metric from these manual checks, ensuring the patterns captured smell-indicative patterns with minimal false positives. This process provided a sanity check that our regex-based baseline was internally consistent. While regex provided a feasible approach for this initial study, future work could explore more advanced contextual methods to improve prevalence estimation fidelity and scalability.

Figure 4.2 visually represents the prevalence of sustainability smells across the dataset. It highlights the relative frequency with which each smell appears. The analysis shows that

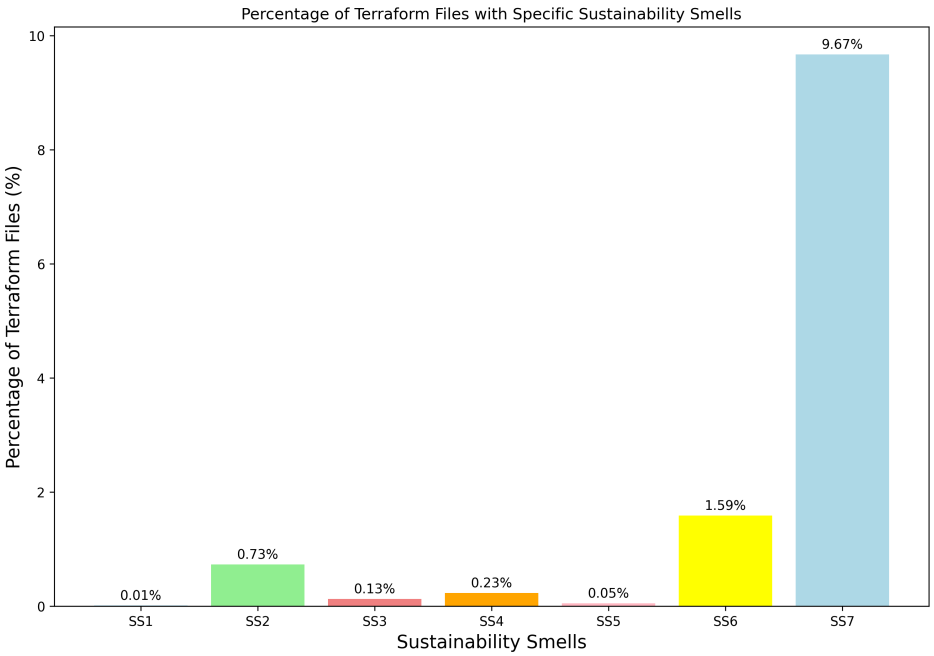

Figure 4.2 Percentage of Terraform Files with Specific Sustainability Smells

SS7 (Non-Modular Configurations) is the most prevalent smell, found in 9.67% of Terraform scripts, indicating a widespread tendency toward non-modular architectures. SS6 (Poor State Management) is the second most common at 1.59%, underscoring recurring challenges in managing Terraform state effectively. Other smells, such as SS2 (Lack of Auto-Scaling) at 0.73% and SS3 (Ignoring Resource Lifecycles) at 0.13%, occur less frequently but still point to areas where infrastructure practices can be improved. SS4 (Excessive Logging) and SS5 (Unoptimized Data Transfers) are rare, with prevalence rates of 0.23% and 0.05%, respectively, while SS1 (Over-Provisioning Resources) appears only marginally at 0.01%.

These results highlight the need for better modularization, improved state management, and broader adherence to established infrastructure best practices. To ensure that the prevalence measurements produced by the regex-based method are reasonable, we performed a limited manual sanity check on a random sample of results. This check was used only to assess internal consistency and plausibility, not to estimate accuracy. It is important to note that the **Prevalence Corpus (Dataset1)** (**386 repositories**; **28,327 Terraform scripts**) is an unlabeled dataset used exclusively for large-scale prevalence measurement and exploratory analysis across all seven sustainability smells and is **not** used for accuracy assessment or formal evaluation.

4.3.1 Limitations of Regex

Regex can only capture predefined syntactic patterns and therefore fails to account for the broader context in which a code fragment is used. As a result, a regex-based approach is generally inaccurate for measuring the prevalence of sustainability smells. This limitation aligns with the context problem, where the same code structure may or may not represent a sustainability issue depending on developer intent or deployment environment.

To re-evaluate the effectiveness of regex-based prevalence approximation, we first refined the regex patterns from our previous study to improve their coverage and precision. The improvements included expanding the set of matching keywords, adding alternative syntax variations, and incorporating multiple pattern formulations for each sustainability smell to capture a broader range of IaC implementations across cloud providers.

We then re-measured the prevalence of sustainability smells using these enhanced patterns on the **Evaluation Benchmark (Dataset2)**. We compute prevalence by calculating the percentage of repositories in which each smell indicator is observed (i.e., at least one match occurs), and compared these values to the ground-truth prevalence for the four labeled smells (SS1, SS3, SS4, SS5). Since the dataset already contained known prevalence values, we directly compared the regex-based results against the ground truth to calculate absolute prevalence error and assess overall performance. This evaluation provided a rigorous measure of how well regex-based rules approximate repository-level prevalence and revealed their persistent limitations in handling contextual variation and developer intent.

4.3.2 Regex-based Prevalence Measurement Example Errors

To better illustrate the limitations of regex-based static matching, we provide concrete examples derived from patterns frequently found in public Terraform repositories. These examples reflect real-world configurations that commonly appear in open-source IaC projects, such as those in the ContainerSolutions Terraform examples [75], AlfonsoF AWS Terraform samples [76], and other public AWS IaC patterns discussed in community anti-pattern analyses [77].

Example 1: False Positive for SS1 (Over-Provisioning)

A frequent pattern in public repositories involves provisioning large instance types directly in the configuration. Regex rules flag any explicit appearance of large instance types as SS1, even when such instance sizes are justified by the workload. Similar patterns appear in Terraform AWS examples such as those in AlfonsoF's public repository [76]. This results in

false positives because regex cannot infer the performance or cost requirements behind the configuration.

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "m5.4xlarge"
}
```

Example 2: False Negative for SS1 (Over-Provisioning)

```
variable "instance_size" {
  default = "m5.large"
}

resource "aws_instance" "api" {
  ami           = "ami-12345678"
  instance_type = var.instance_size
}
```

Public Terraform examples commonly define instance sizes indirectly through variables, hiding the actual value from static matching. Regex rules that match literal instance types will fail to capture this smell indicator, producing *false negatives*. This pattern is prevalent in ContainerSolutions' Terraform example sets [75], where configuration values are abstracted into variables for reusability.

Example 3: False Negative for SS7 (Non-modular Configurations)

```
module "high_performance_cluster" {
  source  = "github.com/org/terraform-aws-modules//cluster"
  workers = 10
}
```

Many public modules abstract their internal resources, making it difficult for regex-based rules to capture architectural smells. Although the internal module logic may create a tightly coupled or non-modular infrastructure, the parent configuration contains no explicit pattern that a regex could match. This behavior is typical in modular AWS patterns found in open-source module libraries [75], resulting in *false negatives* for SS7.

```
resource "aws_s3_bucket_replication_configuration" "replica" {
  role = aws_iam_role.replication_role.arn

  rules {
    destination {
      bucket        = "arn:aws:s3:::analytics-data-eu"
      storage_class = "STANDARD"
    }
    prefix = ""
  }
}
```

Example 4: Missed SS5 (Unoptimized Data Transfers)

Public examples often configure cross-region replication without explicit indicators of data transfer volume. Cross-region replication can significantly increase energy consumption and financial cost, yet regex rules cannot capture this as SS5 unless specific keywords are present. Such replication patterns appear in many AWS public examples and anti-pattern discussions [77]. This results in *false negatives* because the sustainability impact arises from semantics, not syntax.

Example 5: Missed SS7 (Non-modular Configurations)

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "subnet1" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"
}

resource "aws_instance" "web1" {
  subnet_id     = aws_subnet.subnet1.id
  instance_type = "t2.micro"
}
```

A typical monolithic configuration found in various public IaC tutorials. Regex rules cannot infer architectural structure, and therefore cannot capture the SS7 smell indicator even though all resources are coupled in a single file. Similar patterns appear in public AWS IaC

examples and anti-pattern repositories [77]. This results in *false negatives* for modularity-related smells.

4.4 Context-aware Prevalence Estimation With Commit-level Reasoning

Given the limitations of the regex baseline, which provides a reproducible, conservative lower-bound estimate of prevalence but cannot capture developer intent or configuration context. We aim to develop a more accurate approach to measure the prevalence of sustainability smells. Regex-based static matching proved limited because it ignored developer intent and contextual factors in IaC changes. To address this, we leverage commit history data, and we use the LLMs capable of reasoning about intent.

4.4.1 The Context Problem

A key challenge in identifying sustainability smells lies in what we refer to as *The Context Problem*. Regular-expression (regex) based approaches typically operate by statically analyzing IaC scripts. While this approach is straightforward and scalable, it often overlooks the broader development and deployment context in which a configuration decision is made. For example, the mere presence of a large instance type (e.g., `m5.4xlarge`) in a Terraform file may be flagged as over-provisioning, even though it could be justified by the workload's performance requirements or cost-sharing agreements within an organization.

These examples highlight that sustainability smells cannot be reliably interpreted from syntactic patterns alone when the objective is to measure their prevalence in real-world repositories. The presence of a large instance type or a fixed resource count may reflect legitimate workload requirements or organizational constraints that are not visible in the Terraform file itself. When such cases are counted naively, prevalence estimates become inflated due to misclassification. Contextual information derived from version control history provides partial insight into developer intent at the time changes were introduced, allowing certain false positives to be filtered during retrospective analysis. However, even with commit-level context, the resulting measurements should be understood as approximations of prevalence rather than definitive assessments of sustainability.

4.4.2 Developers Intent as Context

Approaching the context problem requires moving beyond static script analysis toward methods that incorporate developer intent and usage scenarios. Commit messages, diffs, and other

development artifacts provide valuable signals of this intent, as they capture the rationale behind changes and the evolution of resource choices.

Commit data appear to be a promising choice because developers modify IaC scripts with specific reasons in mind, and these reasons are often documented in commit history data. By analyzing both the commit diffs and their associated messages, we gain access not only to the technical changes but also to the developer's intent, which is critical for understanding whether a change relates to a sustainability smell. This contextual information directly addresses the shortcomings of regex-based matching, which lacks awareness of purpose or rationale. Developers change IaC scripts for a reason, and commit messages often state that reason. By reading both the diff and the message, we can recover intent cues and assess whether a change plausibly reflects a sustainability smell. LLMs are a good fit here since the nature of commit data is textual, lending itself well to natural language processing techniques and reasoning models.

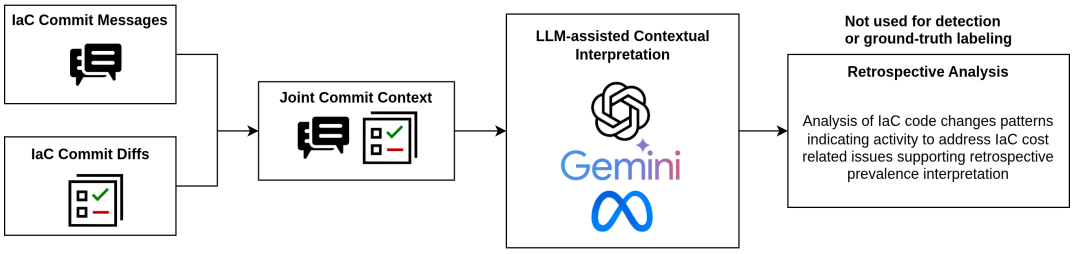

Figure 4.3 How commit history is used for analysis

Commit data are used exclusively to provide retrospective contextualization of mitigation-related actions which can indicate that there was a prevalence of a sustainability smell, see Figure 4.3. Commits are not used to establish ground truth, label smells proactively, or predict future behavior. We apply a multi-stage filtering process to ensure that only relevant commits are analyzed.

First, we restrict analysis to commits that include changes to Terraform configuration files (i.e., files with the `.tf` extension). Commits that do not modify IaC artifacts are excluded, even if their messages reference cost or infrastructure-related concepts.

Second, for commits that modify IaC files, we retain both the commit message and the corresponding code diffs. Commit messages are treated as contextual signals only and are always interpreted in conjunction with the associated IaC changes. Commits whose messages lack sufficient detail are not discarded; instead, their interpretation relies primarily on the observable code modifications.

Third, commits that introduce unrelated changes (e.g., documentation updates, refactoring

of non-IaC files, or mixed commits where IaC changes are incidental) are excluded from our analysis to avoid misattribution of intent.

Finally, all retained commits are analyzed retrospectively and at the repository level. Observations derived from commit analysis are used solely to describe common mitigation strategies and contextual patterns and are not interpreted as comprehensive or definitive representations of developer intent.

In this work, references to developer intent are limited to retrospective interpretation of mitigation-related actions observable in IaC-modifying commits. We do not infer developers' motivations or decision-making processes beyond what can be reasonably contextualized from commit messages and corresponding IaC code changes. In addition, commit messages may be incomplete or ambiguous; to mitigate this, we rely primarily on IaC code changes and use commit messages only as supplementary context rather than as primary evidence of developer intent.

The scripts used for filtration, along with the analyzed commits, can be found in our replication package.[1]

4.5 Evaluation of Regex and LLM methods for Measuring Prevalence

To demonstrate that LLMs provide a more effective NLP-based approach for analyzing commit data, we design a study using the **Evaluation Benchmark (Dataset2)** previously evaluated with regex-based static matching. In this design, we employ the model `llama-3.3-70b-instruct`, prompting it with a few-shot prompt to classify each commit as related or unrelated to sustainability smells by jointly considering both the commit diff and its corresponding message. This setup ensures a fair, one-to-one comparison between regex and LLM-based approaches under identical conditions for the four labeled smells (SS1, SS3, SS4, SS5). Using the ground-truth labels as reference, we compare the prevalence values obtained by both methods to quantify their deviation from the true prevalence numbers. This evaluation provides empirical evidence that LLMs, when exposed to commit-level context and developer intent, offer a more accurate and context-aware alternative for measuring sustainability smell prevalence for these four smells.

Table 4.3 compares regex-based prevalence estimates with the ground-truth dataset. The results reveal substantial discrepancies for certain smells (most notably SS1 and SS5) where regex rules severely underestimate prevalence. This outcome reflects the known limitations of pattern-based heuristics, which struggle to capture semantic intent and contextual cues

[1] `https://github.com/seifkosbar/Sustainability-Smells`

present in IaC scripts. For instance, regex rules may overlook sustainability-relevant configurations when developers use indirect variable references or modularized resource definitions, resulting in high false-negative rates. In addition, because prevalence is aggregated at the repository level, a single missed instance is sufficient to classify an entire repository as negative, amplifying the impact of false negatives. These observations motivate the use of richer contextual signals for prevalence estimation rather than indicating instance-level classification accuracy.

Table 4.3 Comparison of regex-based prevalence and ground truth for selected sustainability smells.

Smell	**Regex Results (%)**	**Ground Truth (%)**	**Absolute Error**
SS1	1.02	66.07	65.05
SS3	2.55	3.57	1.02
SS4	6.63	8.33	1.70
SS5	5.57	27.98	24.41

Table 4.4 Comparison of LLM-predicted prevalence and ground truth for selected sustainability smells.

Smell	**LLM Pred (%)**	**Ground Truth (%)**	**Absolute Error**
SS1	53.55	66.07	12.52
SS3	2.94	3.57	0.63
SS4	6.87	8.33	1.46
SS5	21.94	27.98	6.04

Table 4.4 reports repository-level prevalence estimates produced using LLM-based contextual interpretation. Compared to regex-based heuristics, the absolute deviation from ground-truth prevalence is consistently lower across all evaluated smells. In this setting, absolute error is used to quantify the magnitude of divergence between estimated and ground-truth prevalence distributions at the repository level, rather than to assess instance-level classification performance. This metric is appropriate given that ground truth is available only as aggregated repository-level labels and that the goal of the analysis is comparative prevalence approximation rather than instance-level classification.

The observed reduction in absolute error indicates that incorporating commit-level context allows prevalence estimates to more closely approximate repository-level ground truth under identical aggregation assumptions. We emphasize that these results reflect improved *prevalence approximation*, not instance-level classification accuracy, and should be interpreted accordingly, particularly given the absence of commit- or file-level ground-truth annotations.

For the evaluation conducted in this section, we use the **Evaluation Benchmark (Dataset2)** exclusively. All error rates, and comparative performance results reported in Tables 4.3 and 4.4 are **limited to the four sustainability smells (SS1, SS3, SS4, SS5)** for which repository-level ground truth is available. All prevalence values reported in Tables 4.3 and 4.4 are computed at the **repository level**. A repository is considered to exhibit a sustainability smell if at least one corresponding instance is observed within that repository. This aggregation strategy is inherited from the **Evaluation Benchmark (Dataset2)**, which reports ground-truth cost anti-pattern occurrences at the repository level rather than at the commit or file level. To ensure a fair comparison, both regex-based and LLM-based methods are evaluated using the same repository-level criterion on the same dataset.

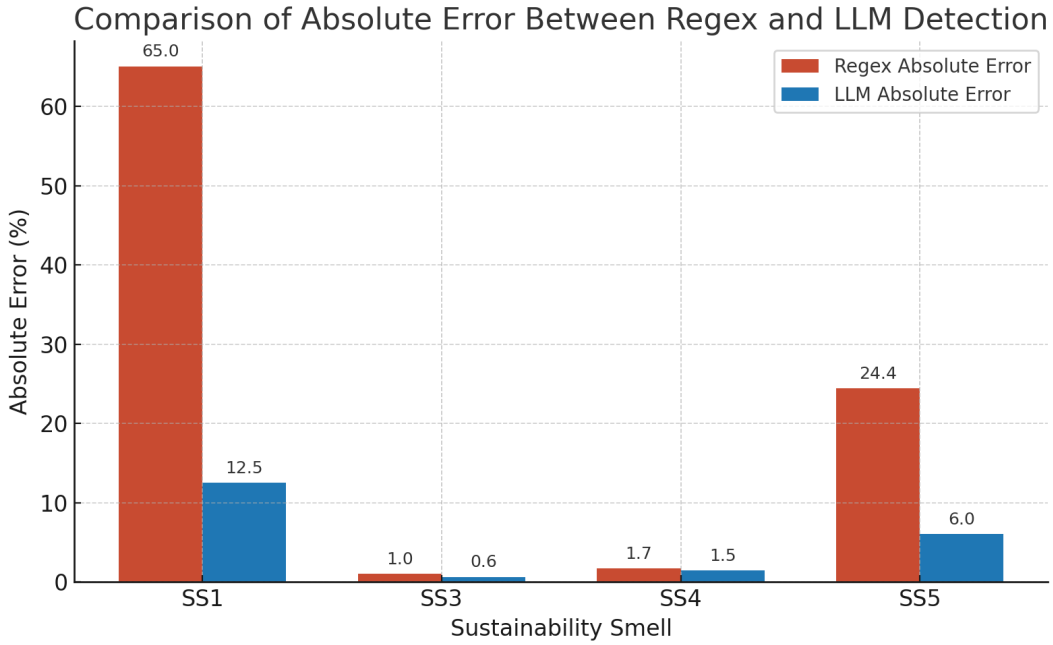

Figure 4.4 Comparison of absolute error between regex and LLM-based prevalence estimation

4.5.1 Failure Modes of Regex and LLMs

Regex failures primarily stem from surface-level pattern matching and manifest in two characteristic modes. The first is **syntactic over-matching**, where generic patterns match code fragments that are not sustainability smells, and the second is **semantic under-matching**, where genuine smells rely on cross-file context or developer reasoning unavailable to static rules. In contrast, LLMs exhibit different failure patterns. It shows **hallucinated inference**, where the model incorrectly infers intent not supported by evidence, **context overweighting**, where commit metadata biases predictions excessively, and **inconsistent application**

of definitions, particularly for smells with nuanced criteria such as SS5. While regex errors are systematic and predictable (stemming from pattern coverage), LLM errors are more stochastic but typically smaller in magnitude. Regex failures primarily stem from surface-level pattern matching and manifest in two characteristic modes. The first is **syntactic over-matching**, where generic patterns match code fragments that are not sustainability smells, and the second is **semantic under-matching**, where genuine smells rely on cross-file context or developer reasoning unavailable to static rules. In contrast, LLMs exhibit different failure patterns, including **hallucinated inference** (incorrectly inferring intent not supported by evidence), **context overweighting** (commit metadata excessively biasing predictions), and **inconsistent application of definitions**, particularly for smells with nuanced criteria such as SS5. While regex errors are systematic and predictable (stemming from pattern coverage), LLM errors are more stochastic but typically smaller in magnitude.

4.6 Re-measuring the Prevalence of Sustainability Smells

Our approach is to measure prevalence through LLM-based commit classification, where commits mined from the **Prevalence Corpus (Dataset1)** (**386 repositories**; **28,327 Terraform scripts**) are analyzed to determine whether they contain one or more sustainability smells across all seven smell categories. For each commit, the model outputs a structured list of inferred smells along with their corresponding categories. These commit-level predictions are then aggregated at the repository level, allowing us to determine which smells are present within each project. Finally, we compute prevalence as the proportion of repositories (or commits, depending on the aggregation level) in which a smell appears relative to the analyzed population. Note that this prevalence measurement covers all seven smells, whereas the accuracy evaluation in Section 4.3 covers only SS1, SS3, SS4, and SS5.

4.6.1 Few-shot Prompt Examples Selection Strategy

When few-shot prompting is used, the example instances are manually selected rather than randomly sampled. The selection is guided by the goal of providing representative coverage of the sustainability smell categories under analysis, rather than optimizing model performance or maximizing accuracy. Examples are chosen to illustrate typical manifestations of each smell category and are reused consistently across repositories and models.

To ensure that commit history is used strictly as contextual evidence and not as a source of new labels or inferences, we formalize the interpretation process through a structured prompt. The prompt explicitly requires joint reasoning over both the commit message and

the corresponding IaC code diff. Code diffs serve as the primary source of evidence, while commit messages are used only to assess intent alignment and confidence. A sustainability smell is considered contextually mitigated only when the code diff exhibits a concrete mitigation signal (e.g., reduced capacity, added lifecycle rules), and the commit message is either aligned or neutral. Commits with contradictory or exploratory intent are explicitly excluded. This design prevents over-reliance on natural language cues and ensures that contextual interpretation remains grounded in observable configuration changes.

Each contextual interpretation is assigned one of two confidence levels: *high* or *low*. A *high* confidence label is assigned when the commit message explicitly aligns with the mitigation signal observed in the IaC diff (e.g., references to rightsizing or cost reduction). A *low* confidence label is assigned when the commit message is neutral or uninformative, but the code diff alone provides strong evidence of mitigation. Commits with ambiguous, contradictory, or exploratory intent are excluded entirely. These confidence labels are later used during verification to prioritize stronger contextual evidence and to assess the robustness of prevalence interpretation.

Illustrative example. Listing 4.3 illustrates how commit-level context is interpreted in practice. The commit message alone suggests a cost-related intent, but the final interpretation depends on the accompanying IaC diff, which provides concrete evidence of mitigation.

Listing 4.3 Example of commit-level contextual interpretation grounded in IaC diffs

```
commit_hash: 3a5c9b8d4e1f2a7b6c8d90123456789abcdef012
commit_message: rightsize API nodes to cut cost during off-peak
commit_diff:
- desired_capacity = 6
+ desired_capacity = 2
```

In this example, the reduction in `desired_capacity` provides direct code-level evidence of mitigation, while the commit message reinforces the interpretation by explicitly referencing rightsizing and cost reduction, resulting in a *high* confidence assessment. If the commit message were neutral, the mitigation would still be considered valid based on the diff alone but would be assigned *low* confidence. Conversely, if the message indicated a temporary or exploratory change (e.g., testing or benchmarking), the commit would be excluded despite the apparent reduction. This conservative interpretation strategy ensures that commit messages supplement, rather than override, code evidence.

Based on the commit in Listing 4.3, the structured output produced by the prompt is shown

in Listing 4.4. The output records the affected smell, the mitigation action, the supporting diff evidence, the intent cues extracted from the commit message, and the assigned confidence level.

Listing 4.4 Structured output produced from commit-level contextual interpretation

```
[
  {
    "commit_hash": "3a5c9b8d4e1f2a7b6c8d90123456789abcdef012",
    "smell": "SS1",
    "action": "removed",
    "evidence": [
      "- desired_capacity = 6",
      "+ desired_capacity = 2"
    ],
    "intent_evidence": [
      "rightsize API nodes",
      "cut cost"
    ],
    "explanation": "The commit message indicates intentional
       rightsizing for cost reduction, and the diff reduces
       desired_capacity from 6 to 2, mitigating over-provisioning (
       SS1).",
    "confidence": "high"
  }
]
```

Model selection strategy and justification:

Throughout this study, we employ different LLMs by various providers to conduct different analytical tasks: We use `llama-3.3-70b-instruct` for the initial regex-vs-LLM comparison, `gpt-4.1-mini` and `llama-4-maverick` for the ablation study, and `gpt-4.1-mini` and `gemini-2.5-flash` for prevalence measurement with cross-model verification. This diversity is intentional and serves multiple methodological purposes.

First, it avoids single-model bias. Relying on a single LLM throughout would risk systematically overestimating or underestimating prevalence due to model-specific reasoning patterns, hallucination tendencies, or prompt sensitivities. By employing multiple models from different vendors (OpenAI, Meta, Google) with distinct architectures and training objectives, we ensure that our findings are not artifacts of a particular model's behavior but reflect more general patterns that emerge across independent reasoning systems.

Second, the model selection aligns with task-specific requirements. For the initial regex-vs-LLM comparison, we selected `llama-3.3-70b-instruct` due to its strong balance of reasoning capability and inference cost, enabling large-scale analysis on the Evaluation Benchmark (Dataset2) without prohibitive computational expense. For the ablation study, we included both `gpt-4.1-mini` (a proprietary frontier model) and `llama-4-maverick` (an open-weight model) to ensure that prompt design insights generalize across model families and are not specific to closed or open ecosystems. For prevalence measurement, we prioritized models with demonstrated reliability in contextual reasoning: `gpt-4.1-mini` for its consistently high precision and `gemini-2.5-flash` for its complementary reasoning behavior, enabling a conservative intersection-based filtering strategy that reduces noise while retaining high-confidence consensus cases.

Third, this approach enables empirical assessment of cross-model agreement as a quality signal. By comparing outputs from models with different strengths and failure modes, we gain insight into which prevalence-related inferences are robust (agreed upon by multiple models) and which are model-specific artifacts. The disagreement set analysis in Section 4.6.4 demonstrates that such cross-model validation is essential for identifying systematic biases (e.g., Gemini's tendency to over-interpret maintenance activities as sustainability improvements) and for establishing confidence in prevalence estimates.

Overall, the diverse model selection strategy strengthens the study's methodological robustness by triangulating findings across independent reasoning systems, ensuring that conclusions are not contingent on the idiosyncrasies of any single model, and providing empirical evidence of the trade-offs inherent in consensus-based filtering.

4.6.2 Ablation Study to Construct The Best Prompt

To identify the most effective prompt configuration, we conduct an ablation study (Figure 4.5) comparing three few-shot prompt designs:

- (1) a *regex-only* prompt encoding syntactic pattern cues without any reference to developer intent.
- (2) an *intent-only* prompt focusing solely on reasoning from commit messages and change rationale.
- (3) a combined *regex + intent* prompt that integrates both sources of information.

Each prompt includes concise, labeled examples to guide the model's reasoning and ensure consistency across runs. For each configuration, we apply the LLM to a subset of the **Preva-**

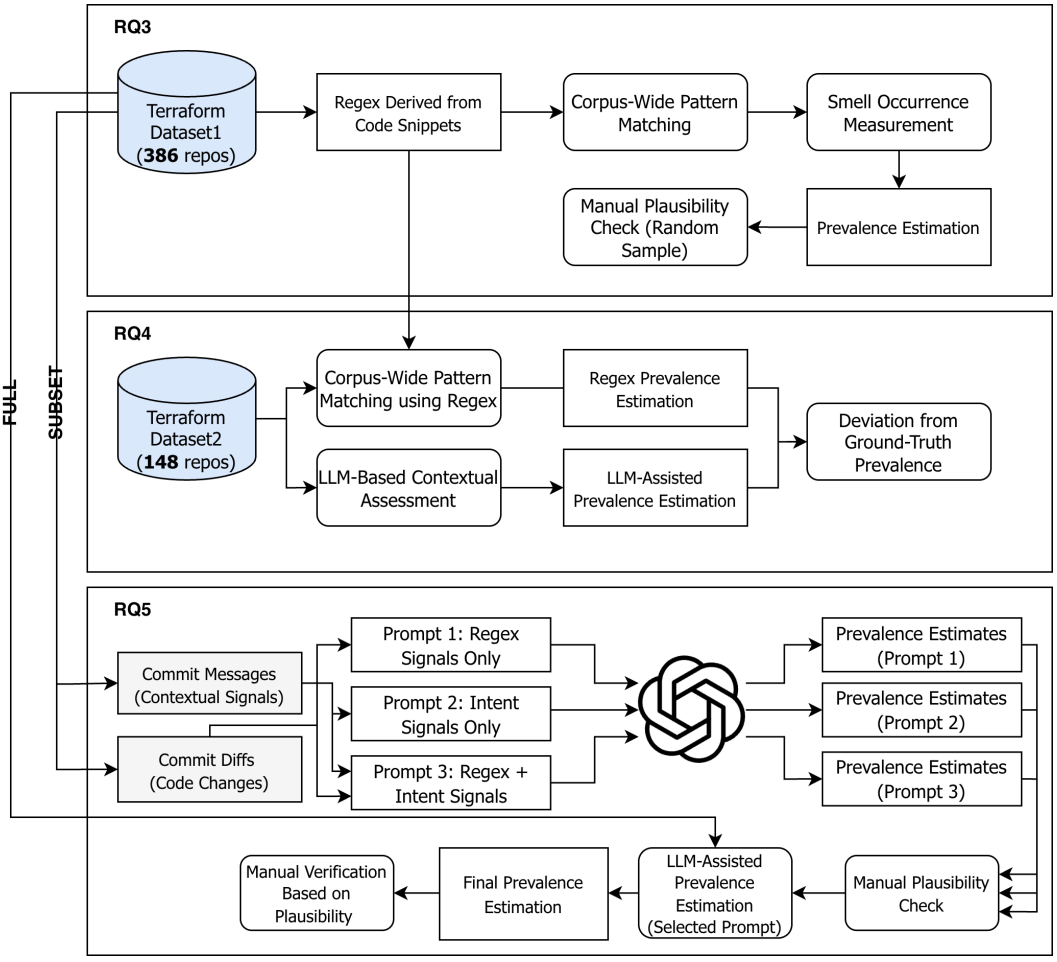

Figure 4.5 Overview of the Approach

lence Corpus (Dataset1) and collect the predicted smell classifications across all seven smells. We then perform stratified sampling across smell categories and prediction outcomes for manual verification, ensuring balanced representation and objective comparison. The manual review results are used to determine the most accurate prompt setting among the three. Once identified, the best-performing prompt is applied to the entire Prevalence Corpus (Dataset1) to produce final prevalence estimates across all seven smells, enabling a direct and fair comparison between the LLM-based and regex-based approaches under identical conditions. We conducted the ablation study using two LLMs which are `gpt-4.1-mini` and `llama-4-maverick`.

After identifying the most accurate prompt, we apply it to the **Prevalence Corpus (Dataset1)** to ensure a direct comparison between the LLM-based approach and the regex-based baseline under identical conditions. For each commit, the best-performing prompt classifies whether a sustainability smell is present, and the resulting predictions are aggregated to compute

prevalence across all smell categories. This setup enables us to assess not only prevalence-estimation error (where ground truth is available) but also the consistency of prevalence estimates across methods.

4.6.3 Ablation Study Results

Table 4.5 presents the ablation study comparing three interpretation settings (*regex only*, *intent only*, and *intent + regex*) across two models. As expected, the regex-only configuration generally reports higher prevalence rates, reflecting its tendency to over-flag commits due to rigid pattern matching and limited contextual understanding, which increases false positives. This behaviour is especially pronounced for smells such as SS1 and SS5, where developers frequently embed configuration values indirectly using variables, locals, or module outputs. Because regex rules treat these cases uniformly regardless of their semantic role, they inflate prevalence by flagging non-smelly changes as sustainability issues.

Conversely, the *intent-only* setting produces lower prevalence values, suggesting that while reasoning over commit messages reduces spurious matches, it may also overlook some valid cases, leading to false negatives. This is most evident in commits with minimal or ambiguous messages (e.g., "fix config" or "update resources"), where intent cues provide insufficient signal. In such cases, the model may fail to connect code-level changes to sustainability-relevant behaviour, yielding underestimates of true prevalence. Moreover, relying solely on intent makes the approach vulnerable to inconsistent commit documentation practices across repositories.

The combined *intent + regex* setting achieves a balance between these two extremes, yielding more stable and realistic prevalence estimates across most smells. This hybrid approach leverages the precision of intent reasoning while retaining the broader coverage of regex cues, resulting in overall improved labeling reliability for prevalence estimation. In practice, commit-level reasoning helps suppress the false positives arising from purely syntactic matching, while regex cues recover cases where intent alone is insufficiently explicit. The resulting estimates more closely align with ground truth, demonstrating that sustainability smells often require both semantic interpretation (why a change was made) and structural evidence (what the configuration contains). This finding highlights that neither intent nor syntax alone is adequate, and that hybrid approaches are more robust for capturing the multifaceted nature of sustainability issues in IaC.

To help decide the best prompt, we applied stratified random sampling (10%), drawing $N = 40 - 42$ commits per prompt variant, with strata proportional across SS1–SS7. Each sampled commit was judged as a true positive (TP), false positive (FP), or uncertain (UNC)

Table 4.5 Ablation Study Results (%) per smell (on sampled subset of Prevalence Corpus (Dataset1), all 7 smells)

Model	Setting	SS1	SS2	SS3	SS4	SS5	SS6	SS7
llama-4-maverick	regex only	11.42	9.18	13.57	10.33	8.94	33.23	40.68
	intent only	6.51	5.73	7.92	6.41	5.12	16.97	23.63
	intent + regex	8.77	7.22	9.36	8.59	7.13	22.18	31.25
gpt-4.1-mini	regex only	12.63	9.47	10.38	12.84	11.67	16.23	7.91
	intent only	7.24	5.58	6.92	7.11	5.43	11.74	4.87
	intent + regex	9.96	7.61	8.24	9.76	8.48	13.84	6.33

by inspecting diff evidence, commit metadata, and the model rationale. Precision is computed as Precision $= \frac{\text{TP}}{\text{TP+FP+UNC}}$, treating UNC conservatively as incorrect.

Table 4.6 presents the ablation results for `gpt-4.1-mini`.

Table 4.6 Ablation results for `gpt-4.1-mini` on 10% fair sample

Prompt	N	TP	FP	UNC	Precision
Prompt1 (regex only)	41	17	15	9	41.4%
Prompt2 (intent only)	42	26	11	5	61.9%
Prompt3 (intent + regex)	40	29	8	3	72.5%

Precision improves monotonically across the ablations: 41.4% (*prompt1*) → 61.9% (*prompt2*) → 72.5% (*prompt3*). This gain is driven primarily by fewer false positives (FP drops from 15 to 11 to 8), showing that combining intent cues with diff-level regex evidence yields the most reliable filtering on this sample.

4.6.4 Verification of Prevalence Measurements

To improve the reliability of prevalence measurements, we refine the set of identified sustainability smells by aggregating outputs from multiple LLMs and retaining only cases of

Table 4.7 Prevalence Results (%) using Intent + Regex Setting (full Prevalence Corpus (Dataset1): 28,327 scripts, all 7 smells)

Model	SS1	SS2	SS3	SS4	SS5	SS6	SS7
`gpt-4.1-mini`	22.92	22.14	23.70	28.12	16.93	34.64	19.27
`gemini-2.5-flash`	38.80	30.99	30.47	34.11	30.73	47.92	61.98

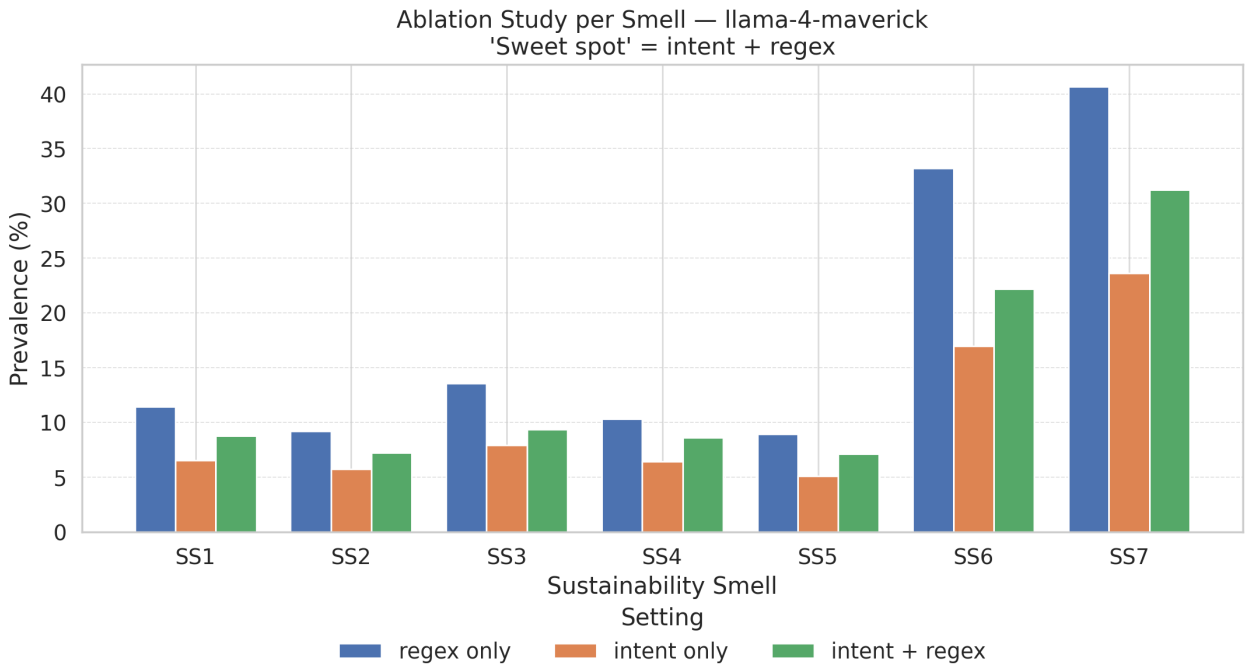

Figure 4.6 Ablation across prompt settings - Llama 4 Maverick

cross-model agreement. Specifically, we combine the results produced by `gpt-4.1-mini` and `gemini-2.5-flash` and retain only commits for which both models assign the same sustainability smell label. This intersection-based filtering is intentionally conservative and is designed to reduce model-specific noise and hallucinations that could otherwise inflate prevalence estimates.

Model outputs are aligned by commit identifiers and compared across smell categories (SS1–SS7). Commits with single-model assignments or conflicting labels are excluded from subsequent analysis. As a result, the retained set consists exclusively of consensus-based contextual interpretations and serves as the basis for prevalence estimation and subsequent verification.

To further assess the soundness of this refinement strategy, we perform an additional manual verification step. Manual verification is conducted exclusively on commits for which both models agree on the assigned smell label, regardless of confidence level. From this consensus-based set, we randomly sample approximately 50% of the available commits, yielding a total of 204 manually inspected commits. Sampling is performed at the repository level to avoid over-representing repositories with large numbers of commits with smell as a stratum.

Each inspected commit is accompanied by a structured JSON record produced by the prompt described in Listing 4.4. This record includes the assigned smell label, mitigation action, supporting diff evidence, intent cues extracted from the commit message, and an associated

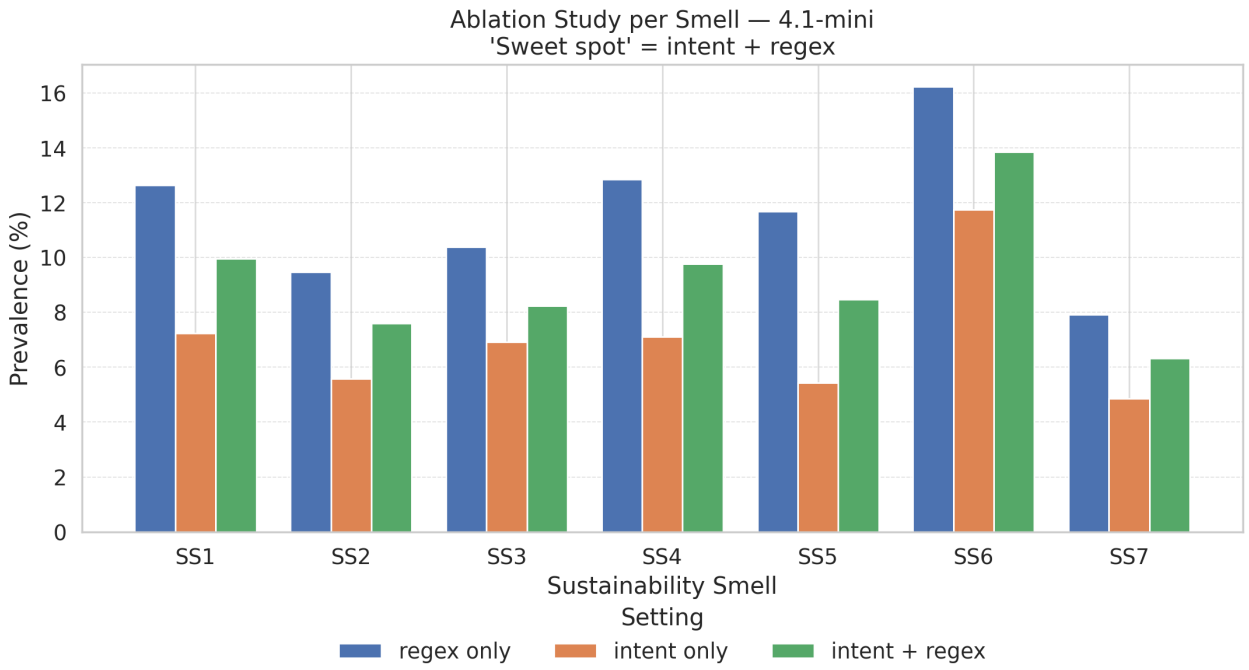

Figure 4.7 Ablation across prompt settings - GPT 4.1-mini

confidence level (high or low). Manual inspection focuses on assessing whether the structured output is plausibly supported by the corresponding IaC diff and commit message.

Although sampling is random, the resulting manual inspection set still provides coverage across all sustainability smell categories and both confidence levels, because commits are randomly selected within each repository and smell stratum rather than from the corpus as a whole. Table 4.8 reports the realized per-smell and per-confidence sample sizes. No commits for which the models disagreed are included in this verification step, as the goal is to assess the internal consistency of consensus-based interpretations rather than to recover missed cases.

This verification process therefore operates in two layers. First, automated cross-model agreement filters the initial set of contextual interpretations. Second, a random subset of these consensus-based instances is manually inspected. This two-layer strategy prioritizes reliability over coverage while maintaining scalability.

Sample size adequacy: The manual inspection of 204 commits, corresponding to approximately half of the available consensus-based instances, provides coverage across all sustainability smells and both confidence levels. This level of sampling is sufficient to surface systematic inconsistencies, recurrent misinterpretations, or confidence-related failure modes,

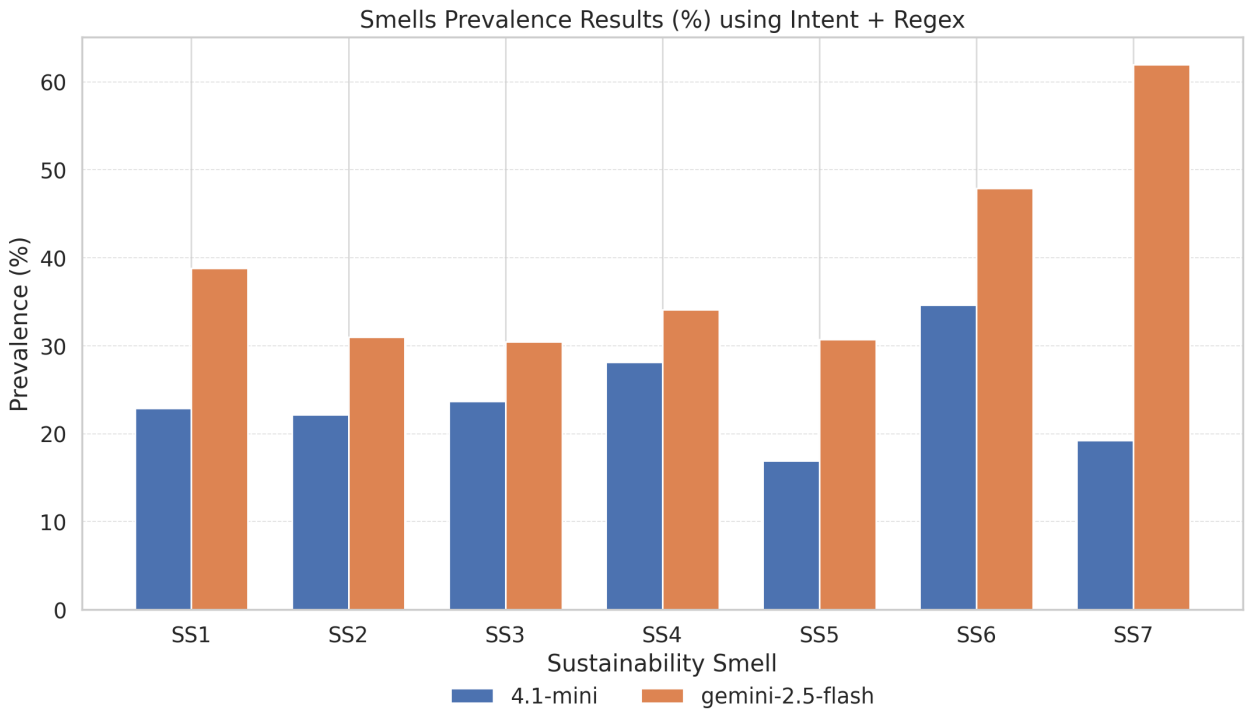

Figure 4.8 Sustainability Smells Prevalence

while remaining feasible for manual analysis. Given that the goal of verification is qualitative assessment of prevalence refinement rather than statistical generalization, this sample size is adequate to support the study's conclusions. The full list of manually inspected commits, along with their structured JSON interpretation records, is made available in the replication package to support transparency and reproducibility.

Estimated precision and confidence intervals: Although manual verification is not used to estimate instance-level classification accuracy or recall, we report indicative precision estimates to summarize the internal consistency of consensus-based interpretations. For each sustainability smell, let n be the number of manually inspected commits and let k be the number judged plausible upon inspection. Precision is computed as $\hat{p} = k/n$. We report an approximate 80% confidence interval using a binomial proportion model with a normal approximation:

$$SE = \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}, \qquad CI_{80\%} = \hat{p} \pm 1.28 \cdot SE.$$

Interval bounds are clipped to $[0, 1]$ when necessary. These confidence intervals are reported solely to characterize uncertainty in the manual verification process and should not be interpreted as instance-level performance, recall, or generalization beyond the verified subset.

Table 4.8 Summary of manual verification sampling (consensus-based commits only)

Stratum	**Manually inspected commits**
SS1 – High confidence	10
SS1 – Low confidence	8
SS2 – High confidence	14
SS2 – Low confidence	12
SS3 – High confidence	12
SS3 – Low confidence	10
SS4 – High confidence	16
SS4 – Low confidence	14
SS5 – High confidence	8
SS5 – Low confidence	6
SS6 – High confidence	24
SS6 – Low confidence	20
SS7 – High confidence	28
SS7 – Low confidence	22
Total inspected commits	**204**

Worked example (80% confidence interval). Using the full manual-inspection sample, $n = 204$ commits were inspected and $k = 124$ were judged plausible. The precision estimate is $\hat{p} = k/n \approx 0.61$. Under a binomial model, the standard error is

$$SE = \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} = \sqrt{\frac{0.61 \cdot 0.39}{204}} \approx 0.034.$$

For an 80% confidence level ($z \approx 1.28$), the confidence interval is

$$CI_{80\%} = \hat{p} \pm z \cdot SE \approx 0.61 \pm 0.044,$$

so the interval is approximately [0.56, 0.65]. Here, the interval is relatively narrow because the sample size is larger ($n = 204$).

Interpretation: if we repeated the same sampling and inspection procedure many times, about 80% of the resulting intervals would contain the true precision for this sample.

We use an 80% confidence level because this manual verification step is exploratory: it is meant to check plausibility and internal consistency, not to support formal statistical inference or performance claims.

Disagreement Set Audit: To assess the reliability of model-specific predictions and to evaluate the effect of the intersection-based filtering strategy, we conducted a targeted audit

of 100 commits drawn from the disagreement set. This set consists of commits for which a sustainability smell interpretation was produced by only one of the two models (`gpt-4.1-mini` or `gemini-2.5-flash`). The quantitative outcomes of this audit are summarized in Table 4.9.

Table 4.9 Audit results for the disagreement set ($N = 100$), broken down by model-reported confidence level

Model	Confidence	Sampled	Plausible	Not plausible	Rate
`gpt-4.1-mini`	High	37	36	1	97%
`gpt-4.1-mini`	Low	13	9	4	69%
`gemini-2.5-flash`	High	34	5	29	15%
`gemini-2.5-flash`	Low	16	6	10	38%

The audit reveals a substantial disparity in plausibility between the two models when considering predictions made in isolation. Unique predictions from `gpt-4.1-mini` exhibit high plausibility for high-confidence predictions (97%) but substantially lower plausibility for low-confidence predictions (69%). The plausible predictions primarily correspond to nuanced sustainability-related changes such as SS1, SS4, and SS7. The false positives (5 cases total: 1 high-confidence, 4 low-confidence) stem from instances where syntactic updates to deprecated rules were over-interpreted as smell removals, despite no substantive change in resource behavior.

In contrast, unique predictions from `gemini-2.5-flash` exhibit poor plausibility even for high-confidence predictions (15%), with low-confidence predictions performing somewhat better but still poorly (38%). This inverted relationship—where low confidence outperforms high confidence—suggests that `Gemini`'s internal confidence calibration does not reliably reflect prediction quality. While its plausible predictions capture several concrete technical mitigations, such as enabling SS6 or SS3, the majority of its implausible predictions arise from misclassifying general code maintenance or cleanup activities (e.g., removing unused resources or refactoring configuration structure) as sustainability-specific improvements, particularly for SS7.

These findings provide empirical support for the intersection-based filtering strategy as a conservative mechanism for prioritizing high-quality contextual interpretations in prevalence estimation. At the same time, the audit highlights an inherent trade-off: enforcing cross-model agreement excludes 45 plausible sustainability smell removals uniquely identified by `gpt-4.1-mini` (36 high-confidence, 9 low-confidence). The confidence breakdown demonstrates that `gpt-4.1-mini`'s high-confidence predictions are highly reliable (97%), while its low-confidence predictions show notably reduced plausibility (69%). In contrast,

`gemini-2.5-flash` exhibits poor plausibility regardless of its reported confidence level (15–38%).

Quantifying the intersection filtering trade-off. The intersection-based filtering strategy introduces a precision-recall trade-off that can be quantified using the disagreement audit results. By requiring cross-model agreement, we **gain** substantial noise reduction: of Gemini's 50 unique predictions, 39 were implausible (78% false positive rate), meaning the intersection filter prevents 39 spurious assignments from inflating prevalence estimates. However, we **lose** valid cases: of GPT's 50 unique predictions, 45 were plausible (90% true positive rate), meaning the filter excludes 45 legitimate sustainability smell cases.

The net effect is a conservative bias that prioritizes precision over recall. For every 45 valid cases excluded, we avoid 39 false positives—a ratio of approximately 1.15:1. This trade-off is justified in the context of prevalence estimation, where systematic over-counting due to model hallucinations would produce misleading ecosystem-wide conclusions. By retaining only consensus-based interpretations, we sacrifice coverage to ensure that reported prevalence values reflect genuine sustainability concerns rather than model-specific artifacts. The excluded GPT-only cases remain accessible in the replication package for alternative analyses that prioritize recall over precision.

All audited commits, verification annotations, and repository-level JSON artifacts are included in the replication package to support transparency and reproducibility.

This behavior is also evident in Figure 4.8, where `gemini-2.5-flash` consistently reports substantially higher prevalence values across nearly all sustainability smell categories compared to `gpt-4.1-mini`. The systematic upward shift suggests a tendency to over-interpret general configuration changes or maintenance activities as sustainability-related signals. This visual discrepancy aligns with the disagreement-set audit results, which show substantially lower plausibility for `gemini-2.5-flash` (15–38%) compared to `gpt-4.1-mini` (69–97%), and further motivates the use of conservative, consensus-based filtering to avoid inflated prevalence estimates.

4.7 Developer Mitigation Strategies

Understanding how developers mitigate them is critical for uncovering recurring best practices. In this study, mitigation strategies are inferred retrospectively from observed configuration changes and their associated commit context, and do not imply guaranteed or explicit developer intent. During contextual prevalence analysis, we prompted the LLMs to generate

an explanation for each identified smell, describing in 1-3 sentences why and how the change resolves it. These explanations provide insight into the developers' mitigation actions and the underlying rationale, offering a strong basis for extracting higher-level themes.

For RQ5, we obtained results from both the `gpt-4.1-mini` and `gemini-2.5-flash` models. Once again, to ensure robustness, we considered only the sustainability smells that were identified by both models. This intersection yields a more reliable set of smells, as it minimizes the likelihood of false positives or false negatives. For each smell, we aggregated the corresponding `gpt-4.1-mini` explanations. We selected the results of this model because it demonstrated stronger consistency and coherence in linking commit message intent to code changes.

4.7.1 LLM-Assisted Thematic Analysis

For each identified sustainability smell, we performed a thematic analysis to uncover recurring mitigation strategies used by developers. In this work, a theme is defined as a recurring, semantically coherent mitigation pattern observed across multiple commits, representing a common developer strategy for addressing a specific sustainability smell. We used thematic analysis, a qualitative research method that identifies, analyzes, and reports patterns within textual data, allowing us to move from individual explanations to generalized insights about how smells are addressed in practice.

We employed `gpt-4.1-mini` for this analysis using a prompt that instructs the model to identify 3–5 distinct themes per smell, summarize the concrete actions developers took, count the number of commits corresponding to each theme, and provide example commits. This ensures that the output captures actionable developer behaviors rather than abstract ideas. To further ensure the quality and reliability of the LLM-generated themes, we manually validated a stratified sample of example commits, confirming that the identified themes accurately reflect the underlying mitigation actions.

4.7.2 Thematic Analysis Results

Our thematic analysis of developer commits identified a range of recurring mitigation strategies for each of the seven sustainability smells. These strategies, summarized in Table C.1 in the Appendix, reveal common patterns in how developers address resource inefficiency, operational risks, and maintainability issues in their IaC configurations.

For **SS1**, the primary mitigation strategy is *right-sizing*. Developers were observed actively downsizing compute instance types to smaller families, reducing the number of resource

replicas, and decreasing allocated storage capacity to better align provisioned infrastructure with actual workload demands. These actions directly combat waste by eliminating excess capacity.

To address **SS2**, developers embrace *automation and dynamic scaling*. Common actions include replacing fixed instance counts with autoscaling groups that can adjust capacity based on demand, adding autoscaling policies tied to performance metrics (e.g., CPU utilization), and deploying Kubernetes-native autoscalers like the Horizontal Pod Autoscaler (HPA) to enable elastic resource allocation.

Mitigations for **SS3** focus on enhancing *operational safety and reducing resource churn*. Developers commonly add `lifecycle` blocks with rules like `create_before_destroy` to ensure zero-downtime updates, `prevent_destroy` to protect critical resources from accidental deletion, and `ignore_changes` to prevent unnecessary and disruptive updates to specific resource attributes.

Regarding **SS4**, mitigation centers on *automated data management and cost optimization*. Developers implement lifecycle rules to automatically transition logs to cheaper, colder storage tiers (e.g., from AWS S3 Standard to Glacier) or to expire and delete them after a defined retention period. In some cases, they remove or disable logging configurations altogether to reduce data volume at the source.

For **SS5**, strategies revolve around *optimizing network architecture for cost and efficiency*. This is achieved by adding VPC endpoints and peering connections to facilitate private data transfers between services, thereby avoiding costly public internet routes. Developers also remove public-facing components like internet gateways when private connectivity is sufficient.

To resolve **SS6**, developers adopt practices that improve *collaboration, security, and consistency*. Key actions include migrating from local state files to remote backends (e.g., S3 or Terraform Cloud), enabling state locking to prevent concurrent modifications, and encrypting state files at rest to protect sensitive infrastructure data.

Finally, for **SS7**, the dominant theme is *improving code structure through refactoring and reuse*. We observed developers refactoring monolithic configurations by extracting inline resources into reusable modules, replacing local module sources with version-controlled remote sources to promote standardization, and eliminating code duplication by consolidating redundant resource definitions.

It is important to note that the analysis of mitigation strategies is conducted retrospectively and serves an explanatory purpose rather than a prescriptive one. By examining how

developers addressed sustainability smells after they were introduced, this section aims to contextualize prevalence measurements with evidence of how such configurations are typically resolved in practice. The extracted strategies reflect historical remediation actions observed in version control history and should not be interpreted as recommendations or automated fixes.

4.7.3 External Validation of Mitigation Strategies

To assess the validity of the mitigation strategies identified in RQ6, we cross-referenced our findings with two external sources: the ground-truth catalog of IaC cost patterns by Bolhuis et al. [78] and official best practice documentation from cloud providers and HashiCorp.

For the sustainability smells that overlap with the cost patterns in Dataset 2 (SS1, SS3, SS4, SS5), the strategies extracted by our model align closely with the "solutions" documented by Bolhuis et al. For example, our finding that developers mitigate **SS1 (Over-provisioning)** by "downsizing instance types" and "reducing resource counts" corresponds directly to the *Expensive Instance* and *Overprovisioned Resources* antipattern solutions. Similarly, the mitigation for **SS4 (Excessive Logging)** via "lifecycle expiration rules" matches the *Object Storage Lifecycle Rules* pattern. This alignment confirms that the LLM is correctly identifying established community practices rather than hallucinating generic advice.

For strategies not covered by the cost catalog (SS2, SS6, SS7), we validated them against official documentation:

- **SS2 (Lack of Auto-Scaling):** The use of autoscaling groups aligns with the *Cost Optimization Pillar* of the AWS Well-Architected Framework, which recommends "matching supply and demand" dynamically to avoid paying for idle capacity.
- **SS6 (Poor State Management):** Migrating to remote backends (e.g., S3 with DynamoDB locking) is the standard recommendation by HashiCorp for collaborative and secure state management.
- **SS7 (Non-Modular Configurations):** Refactoring into reusable modules is a core best practice in the Terraform style guide to promote maintainability and reduce code duplication.

This external validation suggests that the taxonomy of mitigation strategies derived from our thematic analysis reflects technically sound and widely accepted industry standards.

4.8 Threats to Measurement Validity

The objective of this chapter is not to determine whether an infrastructure deployment is sustainable, nor to enable proactive prevention or design-time warnings. Instead, the goal is to **measure the prevalence of sustainability smells** in real-world Terraform repositories as accurately as possible, given the inherent limitations of static IaC artifacts. Prevalence measurements are derived from repository-level ground truth inherited from prior empirical work on cost anti-patterns and refined through contextual interpretation of historical development data. Consequently, the results presented in this chapter should be interpreted as observational measurements of how frequently sustainability smells appear in practice, rather than as assessments of runtime efficiency or environmental impact.

Several measurement validity threats apply to this chapter. First, **construct validity**: some sustainability smells rely on operational or workload context that may not be fully observable in Terraform code. Second, **internal validity**: while the Evaluation Benchmark (Dataset2) provides ground truth for four smells, subjective judgment was required for smell labeling, which introduces annotator bias despite consensus procedures. Third, **external validity**: the selected repositories in both datasets, while diverse, may not represent all IaC practices, especially in private enterprise environments. Fourth, **conclusion validity**: prevalence comparisons depend on precise alignment between smell definitions and labeling outputs; small definitional ambiguities can influence measured error rates. These issues are discussed more extensively in the final chapter, but we include them here for completeness because they directly affect interpretation of the results in this chapter.

Summary of RQ3

RQ3: How frequently do the identified sustainability smells occur in real-world Terraform repositories? We measured smell prevalence across 28,327 Terraform scripts using regex-based matching rules supported by stratified manual validation. The results show strong variation in how sustainability smells manifest in practice. SS7 (Non-Modular Configurations) is the most frequent at 9.67%, followed by SS6 (Poor State Management) at 1.59%. Other smells—SS2 (0.73%), SS3 (0.13%), SS4 (0.23%), SS5 (0.05%), and SS1 (0.01%)—are less common, revealing that issues of modularity and state handling dominate real-world sustainability gaps in IaC code.

Summary of RQ4

RQ4: How accurate are static, rule-based methods at estimating sustainability smell prevalence? To evaluate regex-based prevalence estimation error, we compared repository-level prevalence estimates against a ground-truth dataset. While regex captured some surface-level indicators, it frequently failed for smells requiring semantic or contextual understanding. For SS1 and SS5, the absolute error reached 65.05% and 24.41%, respectively. These results show that static heuristics substantially underestimate repository-level prevalence for context-dependent smells and are insufficient for reliable sustainability measurement.

Summary of RQ5

RQ5: Can commit-level context and reasoning improve sustainability smell prevalence estimation? We introduced a context-aware prevalence estimation pipeline using large language models that reason over commit diffs and messages. The `llama-3.3-70b-instruct` model significantly reduced absolute prevalence error across all evaluated smells, with SS1 dropping from 65.05% to 12.52%. The LLM-based approach incorporated developer intent, cross-file interactions, and semantic relationships missed by regex rules. This demonstrates that context-rich, reasoning-based analysis offers a more accurate and scalable pathway for estimating sustainability smell prevalence in IaC.

Summary of RQ6

RQ6: How Do Developers Mitigate Sustainability Smells? Developers address sustainability smells through a set of recurring, practical mitigation patterns. For issues related to **SS1**, they typically right-size infrastructure by reducing instance sizes, replica counts, or storage allocations to better align capacity with real usage. When dealing with **SS2**, developers transition toward elasticity by introducing autoscaling groups, metric-driven policies, or Kubernetes-native autoscalers that dynamically adjust resources based on demand.

To reduce operational risks associated with **SS3**, developers introduce safer lifecycle rules such as `create_before_destroy`, `prevent_destroy`, and `ignore_changes`, ensuring more stable and predictable updates. In the case of **SS4**, they implement automated data lifecycle policies moving logs to colder storage tiers, expiring old data, or reducing unnecessary logging to control storage overhead.

For **SS5**, developers streamline communication paths by adding VPC endpoints and peering connections or by removing unnecessary public gateways, thus reducing cost and improving efficiency. They further enhance collaboration and security when mitigating **SS6** by migrating to remote state backends, enabling locking, and encrypting state files. Finally, to tackle **SS7**, developers refactor configurations into reusable modules, adopt version-controlled remote sources, and remove duplicated resource definitions, resulting in cleaner and more maintainable IaC.

CHAPTER 5 DISCUSSION, THREATS TO VALIDITY

This chapter has two parts. First, we discuss the main findings across **RQ1**–**RQ5**, synthesizing what the results reveal about sustainability smells in IaC and the practical and methodological implications of our detection and prevalence-estimation approach. Second, we present threats to validity, outlining key internal and external factors that may affect the interpretation and generalizability of the results, along with the mitigation strategies adopted in this thesis.

5.1 Discussion

The analysis of sustainability smells in IaC reveals practical recommendations for practitioners to improve efficiency and resource stewardship in Terraform-managed environments. For **SS1** (Over-Provisioning Resources), practitioners should regularly review resource usage and adjust provisioning to match actual needs. Addressing **SS2** (Lack of Auto-Scaling) by configuring dynamic scaling policies allows infrastructure to better adapt to workload fluctuations. Implementing lifecycle policies (**SS3**) helps avoid redundant resource creation and unintended persistence. For **SS4** (Excessive Logging), setting appropriate retention policies reduces unnecessary storage overhead. Co-locating frequently interacting resources (**SS5**) can limit inefficient data transfers. Improving **SS6** (Poor State Management) through remote backends supports consistency and collaboration. Finally, adopting modular designs (**SS7**) enhances maintainability and reuse across environments.

The dendrogram in Figure 3.4 groups sustainability smells based on shared attributes, revealing clusters of general configuration issues (e.g., **SS3**, **SS4**, **SS6**), demand-related concerns (e.g., **SS1**, **SS2**), and application-specific challenges (e.g., **SS5**). These groupings suggest that sustainability in IaC cannot be addressed uniformly; instead, mitigation strategies must be tailored to the underlying characteristics of each smell category.

Survey responses further highlight the context-dependent nature of sustainability smells in practice. For **SS1**, respondents noted that temporary over-provisioning may be justified during early project stages. **SS2** was widely seen as an opportunity for improvement, particularly in non-production environments. **SS3** and **SS4** were generally acknowledged as problematic, though respondents emphasized the need to balance operational safety, compliance, and sustainability. For **SS5**, practitioners recognized the necessity of cross-regional replication in some cases but encouraged more deliberate design choices to avoid unneces-

sary transfers. Opinions on **SS7** were mixed, reflecting trade-offs between simplicity and long-term maintainability.

Together, these findings reinforce that sustainability in IaC involves balancing project-specific constraints with best practices, rather than enforcing rigid rules. Effective support for sustainable IaC development therefore requires adaptable tools, contextual guidance, and an understanding of developer intent.

Context-aware prevalence interpretation: The results of the context-aware approach provide insight into the methodological trade-offs between purely syntactic heuristics and reasoning-driven contextual interpretation when estimating sustainability smell prevalence. Regex-based heuristics exhibit substantial divergence from repository-level ground truth for several smells, particularly **SS1** and **SS5**. This divergence reflects the limitations of static pattern matching, which cannot account for implicit configurations expressed through variables, modules, or conditional logic. As a result, regex-based approaches tend to underestimate prevalence under repository-level aggregation.

In contrast, LLM-based contextual interpretation produces prevalence estimates that more closely approximate repository-level ground truth under identical aggregation assumptions. By jointly considering commit diffs and associated commit messages, LLMs provide a mechanism for interpreting configuration changes in context, rather than relying on surface-level patterns alone. Importantly, this improvement should not be interpreted as enhanced detection accuracy at the instance level. Instead, it reflects improved *prevalence approximation* achieved by reducing misclassification and ambiguity in retrospective analysis.

Interpretation of RQ5: RQ5 examines whether commit-level context and reasoning improve the contextual interpretation of sustainability smell prevalence. Taken together, the results indicate that commit-level context plays a complementary role in refining prevalence measurements rather than enabling new detections. Commit diffs provide concrete evidence of mitigation-related changes, while commit messages act as secondary signals for intent alignment. This combination allows ambiguous cases to be filtered out and supports conservative, consensus-based interpretations.

The ablation study further clarifies this role. Regex-only interpretation tends to overestimate prevalence due to syntactic overmatching, while intent-only interpretation may miss valid cases when code-level signals are subtle. The combined intent+regex configuration balances these effects, yielding more stable prevalence estimates. Variations across LLMs are expected given differences in reasoning behavior; to address this, we retain only cases of cross-model

agreement, prioritizing reliability over coverage.

Overall, the findings for RQ5 demonstrate that commit-level context improves the *interpretability and robustness* of prevalence measurements by reducing noise and misclassification in retrospective analysis. This contribution aligns with the study's broader goal of producing conservative, transparent prevalence estimates in settings where fine-grained ground truth is unavailable.

Finally, the thematic analysis indicates that developers commonly mitigate sustainability smells through resource optimization, automation, and modularization. Embedding contextual guidance and lightweight reasoning support into IaC workflows could help institutionalize these practices and support more informed decision-making without imposing rigid constraints.

5.2 Implications

This section translates the thesis findings (RQ1–RQ5) into actionable usage scenarios for key stakeholders. The implications should be interpreted as guidance derived from retrospective evidence and empirical patterns observed in open-source Terraform repositories.

5.2.1 Implications for practitioners

First, treat sustainability smells as *risk indicators* rather than hard violations: use them to prioritize review and discussion, especially for context-sensitive smells such as **SS1** and **SS5**. Second, embed lightweight checks into routine workflows (e.g., pull-request reviews) by combining static cues (regex-like detectors) with contextual signals from change history: commit messages and diffs can help distinguish intentional, justified configurations from accidental inefficiencies. Third, operationalize the mitigation themes identified in RQ5 as repeatable practices—for example, right-sizing and elasticity for demand-related smells, lifecycle safeguards for resource churn, and explicit state/logging policies for long-lived infrastructure components.

5.2.2 Implications for tool builders

The results suggest a hybrid tool design. Fast, deterministic static rules provide scalable coverage and a reproducible baseline, while contextual reasoning (e.g., using commit diffs and commit messages) can reduce ambiguity and false positives during retrospective audits. Tool builders can implement multi-stage pipelines that (i) flag candidates with cheap static de-

tectors, (ii) attach structured context (diff snippets, message, file paths, resource metadata), and (iii) apply conservative filters such as cross-model or cross-signal agreement to control noise. Importantly, tools should surface *evidence and rationale* (why a case was flagged, what context supported it) to support developer trust and enable human override.

5.2.3 Implications for researchers

Methodologically, this thesis highlights the value of commit history as contextual data for IaC smell analysis: it can improve prevalence *approximation* and interpretability when fine-grained ground truth is unavailable (RQ4–RQ5). Future work can extend this by (i) studying how smell prevalence varies over time and across domains, (ii) validating whether the same taxonomy and detection strategy transfer to other IaC ecosystems (e.g., CloudFormation, Ansible), and (iii) designing benchmarking protocols that separate instance-level detection accuracy from repository-level prevalence estimation. More broadly, the findings motivate research on conservative measurement designs that prioritize transparency and robustness over maximal coverage.

5.3 Threats to Validity

5.3.1 Internal Validity

There is a risk of selection bias in collecting Terraform scripts from online platforms such as GitHub. Scripts chosen for analysis may not be representative of the entire population of Terraform scripts, potentially leading to skewed results. To mitigate this, we employed a diverse sampling strategy. In addition, the industry reports collected may introduce bias if the selected sources predominantly focus on specific cloud platforms or industries. This bias could impact the identification of energy-intensive practices. To minimize this, we considered a broad range of reports from major cloud providers to encompass multiple perspectives.

One of the challenges in the analysis of IaC scripts is assessing workload demands without runtime execution data. To classify SS1 (Over-Provisioning Resources), we relied on best practices from AWS, Azure, and GCP, which provide guidelines on optimal VM sizing. In future work, we plan to integrate machine learning techniques for more dynamic workload inference.

The reliability of our detection results depends on the accuracy of both regex and LLM-based classifications. Regex detectors may produce false positives or negatives due to syntactic variation, modularization, or indirect resource references that obscure smell indicators. Although

we refined the patterns from our prior study and validated them on a labeled ground-truth dataset, regex methods remain limited by their inability to infer developer intent.

For LLMs, misclassification may arise from prompt sensitivity, reasoning bias, or probabilistic variation across runs. We mitigated these risks by conducting an ablation study to select the most stable prompt configuration, aggregating outputs from multiple models (e.g., `gpt-4.1-mini` and `gemini-2.5-flash`) through intersection filtering, and manually verifying a stratified subset of detections. Nevertheless, LLMs can still exhibit inconsistencies or context misinterpretation, and future replications may observe minor deviations as model architectures evolve.

Another internal threat concerns human judgment in thematic analysis. Although we used LLM-generated explanations as a starting point, all extracted themes and examples were cross-checked by two researchers to minimize subjective interpretation. Despite these precautions, thematic grouping remains interpretive by nature.

The thematic analysis of mitigation strategies relies on LLM-generated summaries followed by manual verification on stratified samples. While this approach improves consistency, it does not eliminate subjective interpretation. Developer rationales not captured in commits may lead to incomplete or imprecise categorizations of mitigation strategies.

The manual verification is performed on instances selected after contextual interpretation and therefore reflects only the set of smells and mitigation actions surfaced by the models. As a result, smell categories that are systematically missed or misclassified by the models cannot be recovered through stratified sampling. This verification process is not intended to measure recall or completeness, but rather to assess the plausibility and internal consistency of the observed interpretations. Consequently, reported improvements should be understood as conditional on the set of model-identified instances and may overestimate robustness for under-represented or ambiguous smells.

Because prevalence is aggregated at the repository level, the reported values may overestimate the proportion of affected repositories relative to instance-level measurements; however, this aggregation is consistently applied across methods and aligns with the structure of the available ground-truth data.

5.3.2 External Validity

The study's findings may not be universally applicable, as they are based on a specific set of Terraform scripts and cloud platforms. This poses a threat to external validity, and we transparently communicate the study's context and limitations, emphasizing its applicability

to Terraform scripts and specified cloud platforms. Additionally, cloud environments are dynamic, and the identified patterns may become obsolete or evolve. We acknowledge this dynamic nature and emphasize the temporal relevance of its findings.

The effectiveness of detection techniques may vary based on specific patterns and chosen scripts, and conclusions about performance might not be generalizable to all sustainability smells or IaC scripts. We acknowledge this limitation, providing insights into the strengths and weaknesses of the techniques within the defined scope. Moreover, the identified code patterns may not encompass all potential manifestations of sustainability smells, as variations in coding styles, project requirements, and deployment scenarios could introduce alternative patterns.

This study analyzes sustainability smells after infrastructure configurations have already been introduced and, in many cases, modified or removed. Git commit history is used to contextualize detected patterns during retrospective prevalence measurement, not to enable prediction or early detection. Consequently, the results describe how sustainability smells appear in historical repositories rather than how they might be prevented in future deployments.

Additionally, the sustainability smells defined in this study serve as recommendations for sustainable, cost-effective cloud infrastructure. However, removing these smells may not necessarily result in significant reductions in energy consumption or cost. The approach remains iterative and adaptive, aiming to refine and expand our understanding of sustainability-related code patterns in Terraform scripts.

It is worth noting that our study primarily relies on sustainability best practices outlined in industry vendor reports. While these recommendations provide valuable insights into optimizing cloud infrastructure, it is important to recognize that vendor guidelines may also reflect commercial incentives, encouraging clients to adopt specific cloud services. To counterbalance this, future work could incorporate community-driven best practices from open-source initiatives, industry forums, and empirical studies analyzing real-world infrastructure deployments. Integrating both vendor and community perspectives would enhance the generalizability of sustainability best practices for IaC.

Our results are derived from open-source Terraform repositories, which may not fully represent industrial-scale IaC systems or proprietary environments with stricter governance, cost monitoring, and sustainability policies. Repository selection bias could influence the observed prevalence of smells and the types of mitigation strategies employed. To counter this, we included projects of varying size, activity, and domain, and relied on smell definitions grounded in vendor-neutral sustainability guidelines. However, results may differ for other IaC tools

such as Ansible, CloudFormation, or Pulumi, or in settings with automated provisioning pipelines and cost optimization frameworks. Furthermore, our focus on cost- and resource-related sustainability excludes other dimensions such as social or process sustainability. The taxonomy and detection methods presented here should therefore be interpreted as a foundation for further exploration rather than an exhaustive characterization of sustainable IaC practices.

Overall, while these limitations constrain generalizability, the use of multiple datasets, cross-model validation, and manual verification provide a balanced and credible assessment of sustainability smell detection and mitigation in Terraform.

CHAPTER 6 CONCLUSION

6.1 Summary of Works

This thesis delivers a comprehensive investigation into sustainability smells in Infrastructure as Code (IaC), combining large-scale empirical analysis, practitioner validation, and the development of context-aware detection techniques. Through a systematic examination of tens of thousands of Terraform scripts, a carefully sampled in-depth analysis, and alignment with sustainability recommendations from major cloud providers, the work establishes a validated taxonomy of IaC sustainability smells that capture practices negatively affecting both cost efficiency and energy usage. Empirical prevalence estimates highlight recurring issues—such as non-modular configurations, inadequate state management, and missing autoscaling—that appear consistently in real-world cloud deployments and represent key opportunities for improving cloud sustainability.

A central contribution of this thesis is demonstrating that sustainability smells cannot be reliably detected through static pattern matching alone. Many issues, including over-provisioning and inefficient data transfer, depend on architectural intent rather than isolated syntactic cues. By leveraging developer commit histories and applying large language models with few-shot reasoning, this thesis introduces a context-aware detection approach that substantially improves accuracy and provides more trustworthy prevalence measurements. Beyond detection, the reasoning-enabled analysis reveals the range of mitigation strategies developers already apply in practice—rightsizing, autoscaling, modularization, and lifecycle management—while also showing how inconsistently these are adopted across infrastructure codebases.

Overall, this thesis positions sustainability in IaC as a socio-technical problem requiring an understanding of both code structure and developer intent. It contributes (1) a validated and practically grounded catalog of sustainability smells, (2) quantitative evidence of their presence in real cloud infrastructure, and (3) a new methodological direction for detecting and interpreting these issues using reasoning-driven models rather than fragile heuristics. These contributions collectively offer a stronger foundation for sustainability-aware cloud engineering and open pathways toward automated detection and remediation systems. Future research should extend these methods to additional IaC tools, integrate runtime and cost telemetry, and build end-to-end pipelines that help teams design, deploy, and maintain cloud infrastructures that are both cost-effective and environmentally responsible.

6.2 Limitations

Despite providing the most extensive investigation to date on sustainability smells in IaC, several limitations remain that should be considered when interpreting the findings and designing future work.

6.2.1 Dataset Coverage and Sampling Bias

The analyses rely on open-source Terraform repositories collected from GitHub, which may not reflect the practices, scale, or constraints of private-sector cloud deployments. Repository selection is influenced by the availability of Terraform code, commit histories, and metadata, creating potential sampling bias toward active and well-documented projects. Although stratified sampling was used to mitigate overrepresentation of large repositories, the resulting dataset may still underrepresent certain domains (e.g., industrial IaC pipelines or regulated cloud environments).

6.2.2 External Development Context

IaC commit messages and diffs may not fully capture the rationale behind all infrastructure changes, as some design decisions and discussions can occur in external artifacts such as issue trackers or pull request review threads. In this work, we do not incorporate such external artifacts. The analysis is intentionally limited to version control data to ensure reproducibility and consistency across repositories, as external discussions are not uniformly available and often require repository-specific access or scraping. Consequently, any rationale that is not reflected in commit messages or observable IaC changes is treated as out of scope.

6.2.3 Incomplete Ground-Truth Coverage

The ground-truth dataset used for evaluating detection accuracy covers only four of the seven defined sustainability smells. This limitation arises because existing public datasets catalog cost-related IaC patterns unevenly. As a result, accuracy for smells such as SS2, SS6, and SS7 cannot be validated against a verified benchmark. While the reasoning-based models show consistent behavior, the absence of comprehensive ground-truth annotations restricts the ability to make definitive accuracy claims for all smell categories.

6.2.4 Limitations of the Antipattern-to-Smell Mapping

The ground truth used in **Dataset2** is inherited from a previously published dataset of IaC cost anti-patterns. Repository-level sustainability smell labels are derived transitively through manual mappings between cost anti-patterns and sustainability smells. This approach has several important limitations:

- **Coverage bias**: Only four of seven smells can be mapped, as Bolhuis et al.'s catalog does not cover all sustainability dimensions (e.g., autoscaling, state management).
- **Severity conflation**: Antipatterns may capture only severe or developer-recognized instances of smells, potentially underestimating milder cases.
- **Temporal validity**: Antipatterns are identified retrospectively; labels may not reflect current repository state.
- **Cost-sustainability gap**: Cost antipatterns optimize for financial efficiency, which correlates with but does not fully capture environmental sustainability.

Despite these limitations, cost antipatterns provide the only publicly available, empirically validated ground truth for sustainability-related IaC inefficiencies at repository scale. The mapping enables comparative accuracy evaluation while acknowledging that it represents a conservative lower bound on actual smell prevalence. Although mappings were reviewed multiple times, this transitive labeling strategy introduces uncertainty that should be considered when interpreting the results.

6.2.5 Limitations of Regex-Based Heuristics

The static pattern-based detectors introduced in this thesis are inherently limited by their inability to capture contextual or semantic information. They assume direct textual manifestations of sustainability issues and therefore underestimate smells involving variable indirection, modularized code, conditional logic, or architectural intent. These limitations, while quantified, mean that all regex-computed prevalence numbers should be interpreted as lower bounds rather than exact measurements.

6.2.6 Reliance on Large Language Models

Although large language models (LLMs) significantly improve contextual detection, they introduce several limitations:

- **Reasoning variability:** Different models may occasionally disagree on smell classifications, indicating instability in interpretive reasoning.
- **Opacity and non-determinism:** LLM decisions are not as easily traceable or reproducible as deterministic regex rules.
- **Hallucinations and over-interpretation:** Even with consensus filtering, LLMs may over-reason about simple diffs or infer intent beyond available evidence.
- **Prompt sensitivity:** Detection quality depends on prompt design; suboptimal instructions can distort reasoning or inflate false positives.
- **Cost and scalability:** Large-scale commit-level LLM analysis is computationally and financially expensive, limiting applicability to extremely large codebases.

6.2.7 Lack of Runtime and Cost Telemetry

A fundamental limitation of this work is its reliance on static IaC artifacts and version control metadata. Many sustainability-relevant properties, including energy efficiency and cost-effectiveness, depend on runtime workload behavior, utilization patterns, and operational policies that cannot be inferred from Terraform scripts or Git commit history alone. Consequently, sustainability smells identified in this study should be interpreted as indicators of potential inefficiency rather than as measurements of actual environmental or economic impact.

6.2.8 Terraform-Centric Scope

This thesis focuses exclusively on Terraform. While Terraform is widely used, other IaC frameworks (e.g., CloudFormation, Pulumi, Ansible, Kubernetes YAML) may exhibit different sustainability patterns. Therefore, the taxonomy and detection methods presented here may require adaptation before generalizing to other IaC ecosystems.

6.3 Future Research

This thesis opens several directions for advancing the study and practice of sustainability in IaC. While the work establishes foundational taxonomies, detection methods, and empirical insights, further research is needed to deepen, generalize, and operationalize these contributions.

Expanding to Additional IaC Technologies

The current analysis focuses exclusively on Terraform. Future research should examine whether the sustainability smells identified here arise similarly in other IaC frameworks such as AWS CloudFormation, Pulumi, Ansible, or Kubernetes YAML. Each tool embodies different abstractions, module systems, and execution semantics, which may give rise to new categories of sustainability smells or alter the manifestation of existing ones. Developing cross-IaC taxonomies and detection mechanisms would broaden the applicability of the findings and help unify sustainability practices across cloud ecosystems.

Integrating Runtime and Cost Telemetry

A key limitation of static IaC analysis is the absence of runtime context. Future work should integrate execution traces, performance metrics, cost telemetry, and workload patterns to distinguish between *potential* and *actual* sustainability issues. This integration would enable more precise assessments of over-provisioning, identify workload-specific inefficiencies, and validate whether infrastructure configurations lead to measurable environmental or financial impacts.

Ground Truth Approximation

The manual labeling process employed in this work provides an expert-informed approximation of sustainability risks based on cost-related infrastructure patterns. While this approach enables large-scale analysis, it does not constitute definitive ground truth regarding sustainability outcomes. Future work integrating runtime monitoring, performance modeling, or telemetry data could provide stronger validation.

Automated Detection and Remediation Pipelines

The context-aware detection methods developed in this thesis highlight the promise of reasoning-driven analysis. An important next step is moving toward automated or semi-automated remediation. Future research can explore:

- Automated refactoring tools that propose or apply fixes for common sustainability smells.
- IDE-integrated assistants that provide real-time sustainability feedback during IaC authoring.

- Policy-driven CI/CD checks that prevent the introduction of sustainability regressions.

Such pipelines would operationalize sustainability awareness and make it an integral part of everyday infrastructure engineering.

Improving Intent Inference and Context Modeling

Commit histories offer a useful but incomplete lens into developer intent. Future work should explore richer sources of context, such as architectural diagrams, discussion threads, issue trackers, design documents, and code review comments. Multimodal LLMs or hybrid reasoning models could combine these signals to more accurately infer the motivations behind configuration decisions and better interpret sustainability trade-offs.

Scaling LLM-Based Analysis

The use of large language models for commit-level reasoning introduces scalability challenges. Future research should investigate:

- More efficient model architectures or distillation approaches tailored to IaC analysis.
- Retrieval-augmented or caching-based systems that reduce redundant inference.
- Batching and parallelization strategies for large repositories.

Advancing the scalability of LLM-supported analysis will be essential for applying these methods in enterprise-scale cloud environments.

Refining and Validating Sustainability Taxonomies

As cloud platforms evolve, new services, resource types, and usage patterns will create additional opportunities for sustainability smells to emerge. Future work should revisit and refine the taxonomy presented here, incorporating input from practitioners, cloud providers, and empirical studies across different domains. Establishing benchmarks and shared datasets for sustainability smells would further support community-driven progress and reproducibility.

Human-Centered and Organizational Perspectives

Finally, sustainability in IaC is as much a human and organizational challenge as a technical one. Future research could investigate:

- How developers perceive sustainability responsibilities and trade-offs.
- How team structures, review processes, and organizational policies influence sustainability outcomes.
- Which educational interventions or tooling practices most effectively promote sustainable IaC design.

By extending the technical foundations developed in this thesis and broadening their scope across tools, contexts, and organizational dimensions, future research can contribute to a more holistic and actionable understanding of sustainable cloud infrastructure engineering.

6.4 Data Availability

All data, code, and materials necessary to reproduce the results of this study are publicly available in our replication package.[1] The replication package includes:

- **Repository Selection:** The complete list of the Terraform repositories used as the Prevalence Corpus (Dataset1), along with scripts to reproduce the selection process from GitHub's public dataset.
- **Detection Patterns:** The full set of regex patterns used for baseline pattern-based detection across all seven sustainability smell categories.
- **LLM Prompts and Configuration:** All prompts used for context-aware detection, including few-shot examples, chain-of-thought templates, and structured output schemas. Model versions (`gpt-4.1-mini`, `gemini-2.5-flash`, `llama-3.3-70b-instruct`, `llama-4-maverick`) and inference settings (temperature, top-p, seed) are fully documented.
- **Manual Audit Data:** All manually inspected commits from the disagreement audit, including structured JSON outputs with plausibility labels, rationales, and confidence levels for transparency and reproducibility.
- **Evaluation Benchmark (Dataset2):** The labeled ground-truth dataset used for accuracy evaluation, with commit-level annotations and references to original sources.

This comprehensive package supports full replication of the detection pipeline, prevalence measurement, and qualitative analysis presented in this work.

[1] `https://github.com/seifkosbar/Sustainability-Smells`

REFERENCES

[1] International Energy Agency (IEA), "Data centres & networks," Mar. 2024. [Online]. Available: https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks

[2] R. Buyya, S. Ilager, and P. Arroba, "Energy-efficiency and sustainability in new generation cloud computing: A vision and directions for integrated management of data centre resources and workloads," 2023. [Online]. Available: https://arxiv.org/abs/2303.10572

[3] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 2010.

[4] M. Guerriero *et al.*, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 580–589.

[5] L. R. de Carvalho and A. P. F. de Araujo, "Performance comparison of terraform and cloudify as multicloud orchestrators," in *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 380–389.

[6] Amazon Web Services (AWS), "Sustainability pillar - aws well-architected framework - sustainability pillar," Oct. 2023. [Online]. Available: https://docs.aws.amazon.com/wellarchitected/latest/sustainability-pillar/sustainability-pillar.html

[7] Google, "Design for environmental sustainability | google cloud." [Online]. Available: https://cloud.google.com/architecture/framework/sustainability

[8] Microsoft, "Cost optimization - microsoft azure well-architected framework," 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/

[9] Google Cloud, "Configuration | cloud deployment manager documentation," 2023. [Online]. Available: https://cloud.google.com/deployment-manager/docs/configuration

[10] HashiCorp, "Terraform documentation," 2023. [Online]. Available: https://developer.hashicorp.com/terraform

[11] Wikimedia Commons, "Incident documentation/20170118-labs," 2017. [Online]. Available: https://wikitech.wikimedia.org/wiki/Incident_documentation/20170118-Labs

[12] R. Hersher, "Amazon and the 150 million dollar typo," 2017. [Online]. Available: https://www.npr.org/sections/thetwo-way/2017/03/03/518322734/amazon-and-the-150-million-typo

[13] C. Neill, P. Laplante, and J. DeFranco, *Antipatterns: Managing Software Organizations and People*. CRC Press, 2011.

[14] V. Andrikopoulos *et al.*, "How to adapt applications for the cloud environment: Challenges and solutions in migrating applications to the cloud," *Computing*, vol. 95, no. 6, pp. 493–535, Dec. 2012.

[15] Various, "Systematic review of infrastructure as code (iac) and gitops for cloud automation and governance," *Research synthesis / systematic review*, 2022. [Online]. Available: https://www.researchgate.net/publication/393288366_Systematic_Review_of_Infrastructure_as_Code_IaC_and_GitOps_for_Cloud_Automation_and_Governance

[16] P. Mell and T. Grance, "The nist definition of cloud computing," National Institute of Standards and Technology, Tech. Rep., 2011.

[17] International Organization for Standardization, "ISO/IEC 22123-1:2023 information technology — cloud computing — part 1: Vocabulary," International Standard, 2023, [Online].

[18] Amazon Web Services, "Our origins," https://aws.amazon.com/about-aws/our-origins/, 2024, [Accessed 28-07-2025].

[19] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[20] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," *Fifth International Joint Conference on INC, IMS and IDC*, pp. 44–51, 2009.

[21] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[22] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, M. Badger, and D. Leaf, "Nist cloud computing reference architecture," National Institute of Standards and Technology, Tech. Rep., 2011.

[23] D. Petcu, "Multi-cloud: expectations and current approaches," *Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds*, pp. 1–6, 2013.

[24] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," *Proceedings of the 10th International Conference on Autonomic Computing*, pp. 23–27, 2013.

[25] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.

[26] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2018.

[27] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 68–73, 2009.

[28] Green-Software-Foundation, "Kepler: Real time energy carbon standards for cloud providers," GitHub project, 2024, https://github.com/Green-Software-Foundation/real-time-cloud.

[29] A. Wittig and M. Wittig, *Amazon Web Services in Action*. Manning Publications Co., 2015.

[30] Microsoft and Forrester, "Infrastructure as code: Devops best practices with microsoft azure," Apr. 2015. [Online]. Available: https://devops.com/wp-content/uploads/2015/04/Microsoft-Forrester-Infra-as-code-TLP_April_2015.pdf

[31] Gartner, "Magic quadrant for cloud infrastructure as a service," 2014. [Online]. Available: https://www.gartner.com/en/documents/2571419

[32] Microsoft, "Security quick links - microsoft azure well-architected framework," Nov. 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/well-architected/security/

[33] Google Cloud, "Google cloud architecture framework: Security, privacy, and compliance," Oct. 2024. [Online]. Available: https://cloud.google.com/architecture/framework/security

[34] Amazon Web Services, "Security pillar - aws well-architected framework," Nov. 2024. [Online]. Available: https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/welcome.html

[35] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*. ACM, 2005, pp. 1–5.

[36] E. Kalliamvakou, L. Singer, G. Gousios, D. M. German, K. Blincoe, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. ACM, 2014, pp. 92–101.

[37] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code - an empirical study," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 45–55.

[38] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[39] Refactoring Guru, "Code smells," Jun. 2024. [Online]. Available: https://refactoring.guru/refactoring/smells

[40] H. Mumtaz, M. Alshayeb, S. Mahmood, and M. Niazi, "A survey on uml model smells detection techniques for software refactoring," *Journal of Software: Evolution and Process*, vol. 31, 2019.

[41] M. Panahandeh, M. Hamdaqa, B. Zamani, and A. Hamou-Lhadj, "Muppit: a method for using proper patterns in model transformations," *Software and Systems Modeling*, vol. 20, pp. 1491–1523, 2021.

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

[43] A. Bangash, "Cost-effective strategies for building energy efficient mobile applications," in *2023 IEEE/ACM 45th International Conference On Software Engineering: Companion Proceedings (ICSE-Companion)*, 2023, pp. 281–285.

[44] A. Gupta, B. Suri, D. Sharma, S. Misra, and L. Fernandez-Sanz, "Code smells analysis for android applications and a solution for less battery consumption," *Scientific Reports*, vol. 14, p. 17683, 2024.

[45] B. Zhang, P. Liang, X. Zhou, X. Zhou, D. Lo, Q. Feng, Z. Li, and L. Li, "A comprehensive evaluation of parameter-efficient fine-tuning on method-level code smell detection," *arXiv preprint arXiv:2412.13801*, 2024.

[46] A. Ho, A. M. T. Bui, P. T. Nguyen, A. Di Salle, and B. Le, "Ensesmells: Deep ensemble and programming language models for automated code smells detection," *arXiv preprint arXiv:2502.05012*, 2025.

[47] H. Liu, Y. Zhang, Z. Zhao, and X. Xia, "Prompt learning for multi-label code smell detection," *arXiv preprint arXiv:2402.10398*, 2024.

[48] M. Guerriero *et al.*, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 580–589.

[49] J. Steffens *et al.*, "Code smells in infrastructure as code," in *Proceedings of the International Conference on Software Quality, Reliability and Security (QUATIC/ICSQ)*, 2018. [Online]. Available: https://swc.rwth-aachen.de/docs/2018_QUATIC_Steffens.pdf

[50] A. Rahman *et al.*, "Security smells in infrastructure as code scripts (slic)," *Empirical Software Engineering / Conference Proceedings*, 2021. [Online]. Available: https://akondrahman.github.io/files/papers/slic-sp2021.pdf

[51] J. Corbin and A. Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, 2014.

[52] Z. Botev and A. Ridder, "Variance reduction in monte carlo via the median of adaptive weights," *Journal of the American Statistical Association*, vol. 112, no. 517, pp. 1318–1329, 2017.

[53] A. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Thousand Oaks, CA: Sage Publications, 1998.

[54] M. B. Miles, A. M. Huberman, and J. Saldaña, *Qualitative Data Analysis: A Methods Sourcebook*, 4th ed. Thousand Oaks, CA: Sage Publications, 2020.

[55] A. Horri and G. Dastghaibyfard, "A novel cost based model for energy consumption in cloud computing," *The Scientific World Journal*, vol. 2015, pp. 1–10, 2015.

[56] A. Mosa *et al.*, "Optimizing virtual machine placement for energy and sla," *Journal of Cloud Computing (Springer / BMC)*, 2016. [Online]. Available: https://pure.manchester.ac.uk/ws/files/50446387/art_10_1186_s13677_016_0067_7.pdf

[57] Amazon Web Services, "Right sizing: Provisioning instances to match workloads," https://d1.awsstatic.com/whitepapers/cost-optimization-right-sizing.pdf, 2020, accessed 2025.

[58] ——, "Right sizing is an ongoing process," https://docs.aws.amazon.com/whitepapers/latest/cost-optimization-right-sizing/right-sizing-ongoing-process.html, 2020, accessed 2025.

[59] Various, "Cloud cost optimization methodologies for cloud migrations," *ResearchGate Preprint*, 2025, accessed 2025. [Online]. Available: https://www.researchgate.net/publication/386333614_Cloud_Cost_Optimization_Methodologies_for_Cloud_Migrations

[60] IEA 4E, "Data centre energy use: Critical review of models and results," https://www.iea-4e.org/wp-content/uploads/2025/05/Data-Centre-Energy-Use-Critical-Review-of-Models-and-Results.pdf, 2025, accessed 2025.

[61] S. Alharthi *et al.*, "Auto-scaling techniques in cloud computing: A comprehensive review," *Journal Indexed in PubMed Central*, 2024, accessed 2025. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC11398277/

[62] HashiCorp Terraform, "lifecycle meta-argument reference," https://developer.hashicorp.com/terraform/language/meta-arguments/lifecycle, 2025, accessed 2025.

[63] HashiCorp Terraform Tutorials, "Manage resource lifecycle," https://developer.hashicorp.com/terraform/tutorials/state/resource-lifecycle, 2025, accessed 2025.

[64] AWS Observability Blog, "Optimize log retention to reduce cloudwatch costs," https://aws.amazon.com/blogs/mt/optimizing-log-retention-in-cloudwatch/, 2023, accessed 2025.

[65] Amazon Web Services, "Amazon cloudwatch pricing," https://aws.amazon.com/cloudwatch/pricing/, 2024, accessed 2025.

[66] ——, "Managing cloudwatch costs," https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch_costs.html, 2024, accessed 2025.

[67] Various, "A survey on data center energy consumption and sustainability," *IEEE Access*, 2023, accessed 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10136492

[68] Spacelift.io, "Terraform state: Everything you need to know," https://spacelift.io/blog/terraform-state, 2023, accessed 2025.

[69] HashiCorp Terraform, "State locking," https://developer.hashicorp.com/terraform/language/backend/state#state-locking, 2025, accessed 2025.

[70] ——, "Module documentation," https://developer.hashicorp.com/terraform/language/modules, 2025, accessed 2025.

[71] F. e. a. Rahman, "Infrastructure as code: A study of modularity and maintainability," *IEEE International Conference on Software Maintenance*, 2021, accessed 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9576770

[72] H. Hexmoor, *Computational Network Science: An Algorithmic Approach*. Morgan Kaufmann, 2014.

[73] F. Nielsen, "Hierarchical clustering," in *Undergraduate Topics in Computer Science*. Springer International Publishing, 2016, pp. 195–211.

[74] K. Bolhuis, A. Neamt, D. Feitosa, and V. Andrikopoulos, "IaC cost patterns," 2025.

[75] ContainerSolutions, "terraform-examples," https://github.com/ContainerSolutions/terraform-examples, 2020, public Terraform examples repository demonstrating canonical Terraform configurations and patterns.

[76] AlfonsoF, "terraform-aws-examples," https://github.com/alfonsof/terraform-aws-examples, 2019, terraform examples on AWS covering basic to intermediate infrastructure patterns.

[77] Reaver, "Terraform anti-patterns: Practices to steer clear of," https://reaverops.medium.com/terraform-anti-patterns-practices-to-steer-clear-of-b7ce2784e85d, 2021, discussion of common Terraform anti-patterns and best practices to avoid them.

[78] K. Bolhuis, D. Feitosa, and V. Andrikopoulos, "A catalog of cost patterns and antipatterns for infrastructure as code," in *Proceedings of the 2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2024, pp. 399–406.

APPENDIX A EXAMPLE CODES DERIVED FROM AWS, AZURE, AND GCP SUSTAINABILITY REPORTS

This appendix presents the coding framework derived from major cloud provider sustainability reports. Each code represents a sustainability best practice identified across AWS, Azure, and GCP documentation.

Table A.1 Sustainability Codes from Cloud Provider Reports

Code	**Source**	**Example 1**	**Example 2**
Time-Consuming Resources	AWS, Azure	**AWS:** "Optimize your code that runs within different components of your architecture to minimize resource usage while maximizing performance".	**Azure:** "Optimize component costs. Regularly remove or optimize legacy, unneeded, and underutilized workload components, including application features, platform features, and resources".
Auto-Scaling	AWS, Azure	**AWS:** "Use elasticity of the cloud and scale your infrastructure dynamically to match supply of cloud resources to demand and avoid overprovisioned capacity in your workload".	**Azure:** "Optimize scaling costs. Evaluate alternative scaling within your scale units".

Continued on next page

Table A.1 – *Continued from previous page*

Code	**Source**	**Example 1**	**Example 2**
Remove Low Usage Resources	AWS, GCP	**AWS:** "Use efficient software and architecture patterns such as queue-driven to maintain consistent high utilization of deployed resources".	**GCP:** "Idle resources incur unnecessary costs and emissions".
Minimize Hardware Amount	AWS, GCP	**AWS:** "Use the minimum amount of hardware for your workload to efficiently meet your business needs".	**GCP:** "Consider serverless options for workloads that don't need VMs. These managed services often optimize resource usage automatically, reducing costs and carbon footprint".
Optimize Hardware Accelerators	AWS	**AWS:** "Optimize your use of accelerated computing instances to reduce the physical infrastructure demands of your workload".	**N/A**
Minimize Network Traffic	AWS, GCP	**AWS:** "Use shared file systems or object storage to access common data and minimize the total networking resources required to support data movement for your workload".	**GCP:** "Minimize the total networking resources required to support data movement for your workload".

Continued on next page

Table A.1 – *Continued from previous page*

Code	**Source**	**Example 1**	**Example 2**
Unnecessary Backups	AWS	**AWS:** "Avoid backing up data that has no business value to minimize storage resources requirements for your workload".	**N/A**
Upgrade for Sustainability	AWS, Azure	**AWS:** "Continually monitor and use new instance types to take advantage of energy efficiency improvements".	**Azure:** "Optimize environment costs. Align spending to prioritize preproduction, production, operations, and disaster recovery environments. For each environment, consider the required availability, licensing, operating hours and conditions, and security".
Redundant Data	AWS	**AWS:** "Remove unneeded or redundant data to minimize the storage resources required to store your datasets".	**N/A**

Continued on next page

Table A.1 – *Continued from previous page*

Code	**Source**	**Example 1**	**Example 2**
State Management	GCP	**GCP:** "The Terraform state file is critical for maintaining the mapping between Terraform configuration and Google Cloud resources. Corruption can lead to major infrastructure problems".	**N/A**

APPENDIX B DEFINED SUSTAINABILITY SMELLS

This appendix catalogues the seven sustainability smells identified in Infrastructure as Code configurations, organized by category with illustrative code samples.

Table B.1 Sustainability Smells in Infrastructure as Code

Sustainability Smell	**Category**	**Code Sample**
SS1: Over-Provisioning Resources	2	`resource "azurerm_virtual_machine" "inefficient_vm" { name = "example-vm" location = "East US" vm_size = "Standard_D16s_v3" # Overprovisioned }`
SS2: Lack of Auto-Scaling	2	`resource "aws_instance" "app" { count = 5 # Fixed number ami = "ami-0c55b159cbfafe1f0" instance_type = "m5.2xlarge" }`
SS3: Ignoring Resource Lifecycles	1	`resource "azurerm_managed_disk" "example" { name = "example-disk" storage_account_type = "Standard_LRS" create_option = "Empty" lifecycle { create_before_destroy = false } }`

Continued on next page

Table B.1 – *Continued from previous page*

Sustainability Smell	**Category**	**Code Sample**
SS4: Excessive Logging	1	`resource "aws_cloudwatch_log_group" "detailed_logs" { name = "detailed-log-group" retention_in_days = 365 # Long retention period ... }`
SS5: Unoptimized Data Transfers	3	`resource "google_compute_instance" "example" { name = "example-instance" machine_type = "n1-standard-1" zone = "us-west1-a" # Data frequently transferred # to another region }`
SS6: Poor State Management	1	`terraform { backend "gcs" { bucket = "my-terraform-state" prefix = "network" } }`
SS7: Non-Modular Configurations	1	`resource "google_compute_network" "main" { ... } resource "google_compute_subnetwork" "example" { ... } # Many more resources follow...`

APPENDIX C THEMATIC ANALYSIS RESULTS

This appendix presents the complete results of the thematic analysis conducted on refactoring patterns for each sustainability smell. The themes represent common approaches developers use to address each smell type.

Table C.1 Thematic Analysis of Sustainability Smell Refactoring Patterns

Smell	Theme	Description
SS1: Over-Provisioning Resources		
SS1	Downsize Instance Types	Developers reduce the size or class of compute instances to better match actual resource needs, removing over-provisioning by selecting smaller or less powerful instance types.
SS1	Reduce Resource Counts	Developers decrease the number of instances, replicas, or other resource counts to eliminate excess capacity and reduce over-provisioning.
SS1	Downsize Storage Capacity	Developers reduce the size of storage volumes, IOPS, or other storage-related resources to eliminate excess storage capacity and remove over-provisioning.
SS2: Lack of Auto-Scaling		
SS2	Replace Fixed Capacity with Autoscaling Group	Developers replace fixed instance counts or desired capacities with autoscaling groups configured with min_size, max_size, and desired_capacity parameters to enable dynamic scaling.

Continued on next page

Table C.1 – *Continued from previous page*

Smell	Theme	Description
SS2	Add Autoscaling Policies and Targets	Developers add autoscaling policies, targets, and scaling schedules such as aws_appautoscaling_target, aws_appautoscaling_policy, and CloudWatch alarms to enable dynamic scaling based on metrics.
SS2	Deploy Cluster Autoscaler or Kubernetes Autoscalers	Developers deploy cluster autoscaler components, horizontal pod autoscalers (HPA), vertical pod autoscalers (VPA), or Kubernetes event-driven autoscalers (KEDA) to enable autoscaling in Kubernetes environments.
SS3: Ignoring Resource Lifecycles		
SS3	Add create_before_destroy Lifecycle Rule	Developers add lifecycle blocks with create_before_destroy set to true to ensure new resources are created before old ones are destroyed, preventing downtime and accidental deletions.
SS3	Add prevent_destroy Lifecycle Rule	Developers add lifecycle blocks with prevent_destroy set to true to protect resources from accidental deletion, enhancing resource lifecycle safety.
SS3	Add ignore_changes Lifecycle Rule	Developers add lifecycle blocks with ignore_changes to specific resource attributes to prevent unnecessary resource updates and reduce resource churn.
SS4: Excessive Logging		

Continued on next page

Table C.1 – *Continued from previous page*

Smell	Theme	Description
SS4	Add or Update Lifecycle Expiration Rules	Developers add or modify lifecycle expiration rules on storage buckets or log groups to automatically delete logs or objects after a specified retention period, thereby reducing excessive log retention.
SS4	Add or Update Lifecycle Transition Rules	Developers add or modify lifecycle transition rules to move logs or data to cheaper or colder storage classes after a certain period, reducing storage costs and the impact of excessive logging.
SS4	Remove or Disable Logging Resources	Developers remove or disable logging resources such as log groups, log buckets, or logging configurations to reduce the volume of logs generated or retained, thereby eliminating excessive logging.
SS5: Unoptimized Data Transfers		
SS5	Add VPC Endpoints	Developers add VPC endpoint resources to enable private connectivity to AWS services, reducing cross-region or public data transfers and thus mitigating unoptimized data transfers.
SS5	Add VPC Peering Connections	Developers add VPC peering connection resources to establish private connectivity between VPCs, reducing cross-region or public data transfers and mitigating unoptimized data transfers.
SS5	Remove Public or External Data Transfer Paths	Developers remove public-facing load balancers, internet gateways, or open egress rules to eliminate unoptimized public or cross-region data transfers, improving data transfer efficiency.
SS6: Poor State Management		
SS6	Use Remote Backend	Developers replace local Terraform state storage with remote backends such as S3, AzureRM, GCS, or Terraform Cloud to improve state management.

Continued on next page

Table C.1 – *Continued from previous page*

Smell	Theme	Description
SS6	Enable State Locking	Developers add or configure state locking mechanisms, typically using DynamoDB tables, to prevent concurrent modifications and improve state consistency.
SS6	Enable Encryption for State Storage	Developers configure server-side encryption, often using AWS KMS keys or AES256, for S3 buckets or DynamoDB tables to secure Terraform state data.
SS7: Non-Modular Configurations		
SS7	Refactor Inline Resources into Reusable Modules	Developers extract inline resource definitions from large flat Terraform configurations and move them into reusable modules, replacing the inline resources with module calls to improve modularity and maintainability.
SS7	Replace Local Module Sources with Remote Module Sources	Developers replace local relative module source paths with remote module sources such as git URLs or Terraform Registry modules to promote reuse and reduce duplication, thereby improving modularity.
SS7	Remove Duplicate or Redundant Resources	Developers identify and remove duplicate or redundant resource definitions and module blocks, consolidating them into single reusable modules or data sources to reduce duplication and improve modularity.