Déjà Vu? Redundant Dependencies in Terraform

Carlos Felgueiras*
INESC-ID & IST, University of Lisbon
Lisbon, Portugal
carlosfelgueiras@tecnico.ulisboa.pt

Luís W. Barbosa*
Faculty of Engineering,
University of Porto
Porto, Portugal
luiswbarbosa@gmail.com

Nuno Saavedra
INESC-ID & IST, University of Lisbon
Lisbon, Portugal
nuno.saavedra@tecnico.ulisboa.pt

Alexandra Mendes
INESC TEC, Faculty of Engineering,
University of Porto
Porto, Portugal
alexandra@archimendes.com

João F. Ferreira
INESC-ID and Faculty of Engineering,
University of Porto
Porto, Portugal
joao@joaoff.com

Abstract

Infrastructure as Code (IaC) enables practitioners to provision cloud infrastructure through declarative specifications. In Terraform, developers can define explicit dependencies to enforce execution order when implicit dependencies are insufficient. However, developers may explicitly declare dependencies that are already captured by expression references. Such redundant dependencies add unnecessary complexity and, more importantly, may cause Terraform to produce overly conservative plans and unnecessarily constrain the order in which infrastructure objects are provisioned.

To enable this study, we develop `depends_off`, a static analysis tool for detecting redundant dependencies in Terraform configurations, and use it to analyze 76,296 repositories comprising 226,662 modules. We investigate the prevalence and characteristics of redundant dependencies and their impact on Terraform dependency graphs. We detect 89,570 redundant dependencies, affecting 17% of repositories and 10% of modules. Redundancy is strongly associated with particular resource types and providers, but not with repository-level characteristics such as size, age, or activity. Moreover, removing possibly redundant dependencies reduces the critical path in 8.6% of the cases we could measure, with a median reduction of 14.3% among affected graphs. These findings establish redundant dependencies as a recurring and potentially consequential IaC quality issue and motivate their automated detection as part of Terraform development and maintenance.

CCS Concepts

• **Software and its engineering → Software configuration management and version control systems**.

Keywords

Infrastructure as Code, Terraform, Dependency Analysis, Redundant Dependencies, Static Analysis, Cloud Infrastructure

*Both authors contributed equally to this research.

1 Introduction

Infrastructure as Code (IaC) enables developers and operators to define, provision, and evolve infrastructure through machine-readable configuration files. Among IaC technologies, Terraform has emerged as one of the most widely adopted tools for provisioning cloud infrastructure [27]. Given a declarative specification of the desired infrastructure, Terraform determines how to provision it by constructing a dependency graph between infrastructure objects. This graph establishes the order in which resources can be created, modified, and destroyed, while allowing independent operations to execute in parallel. Consequently, the dependencies in a Terraform configuration play a central role in the provisioning process, influencing how Terraform plans and executes infrastructure changes.

Terraform derives most dependencies automatically from references between objects. For example, when the configuration of one resource refers to an attribute of another resource, Terraform infers that the former depends on the latter. There are, however, dependencies that Terraform cannot infer from such references. For these cases, the language provides the `depends_on` meta-argument, which allows developers to explicitly introduce dependencies into the graph. Developers may, however, introduce explicit dependencies that Terraform can already infer from the configuration. We refer to these as *redundant dependencies*. Such dependencies add unnecessary complexity and may overly constrain Terraform's provisioning process, particularly in large configurations.

Although previous research has extensively investigated IaC quality, including security smells [4, 16, 19, 22, 24], configuration defects [3, 9, 17], automated repair [23], and practitioner-reported challenges [7], comparatively little attention has been paid to the dependency structures underlying IaC configurations. In particular, to the best of our knowledge, there is no large-scale empirical evidence about the use of redundant explicit dependencies in Terraform. It is therefore unclear whether redundant dependencies are merely occasional mistakes or a recurring phenomenon in real-world infrastructure code, what forms these redundancies take, and which Terraform constructs and infrastructure resources are most commonly involved. Addressing these questions is important for practitioners seeking to maintain concise and predictable infrastructure specifications and for researchers and tool developers designing automated analyzes for IaC.

In this paper, we address this gap through the first large-scale empirical study of redundant dependencies in Terraform configurations. To enable the study, we develop `depends_off`, an open-source static analysis tool for identifying redundant dependencies in Terraform. We apply `depends_off` to 76,296 repositories comprising 226,662 Terraform modules, detecting 89,570 redundant dependencies. We find that redundancy affects 17% of repositories and 10% of modules, and is strongly associated with particular resource types and providers, but not with repository-level characteristics. Removing possibly redundant dependencies reduced the critical path in 8.6% of the cases we could measure; among affected graphs, the median reduction in critical-path length and increase in mean parallelism were both 14.3%. These results show that redundant dependencies are a recurring IaC quality issue that can unnecessarily constrain Terraform's provisioning process.

In summary, this paper makes the following contributions:

- We formulate and empirically characterize the problem of *redundant dependencies* in Terraform.
- We present `depends_off`, an open-source static analysis tool for detecting redundant dependencies, including those spanning module boundaries.
- We conduct the first large-scale empirical study of redundant Terraform dependencies, quantifying their prevalence, characteristics, and impact on dependency graphs.

All data and scripts required to reproduce the experiments in this paper are available in our replication package:

https://doi.org/10.5281/zenodo.22004208

2 Redundant Dependency Detection

In this section, we define redundant dependencies in Terraform and describe our approach for automatically detecting them.

2.1 Redundant Dependencies in Terraform

Terraform determines the order in which infrastructure objects are provisioned by analyzing dependencies between them. Most dependencies are inferred automatically from the configuration structure and *expression references*: when the configuration of one object references an attribute of another, Terraform establishes the corresponding dependency [10]. We call those dependencies *implicit dependencies*. When a dependency cannot be inferred from such references, developers can explicitly declare it using the `depends_on` meta-argument, creating an *explicit dependency*.

We define a *redundant dependency* as an explicit dependency that is already implied by other dependencies in the configuration, either through implicit references or through other explicit dependencies. In Listing 1, the `aws_subnet.main` resource refers to the `id` of `aws_vpc.main`, which creates an implicit reference to the resource, but then creates a (redundant) explicit dependency on the same resource.

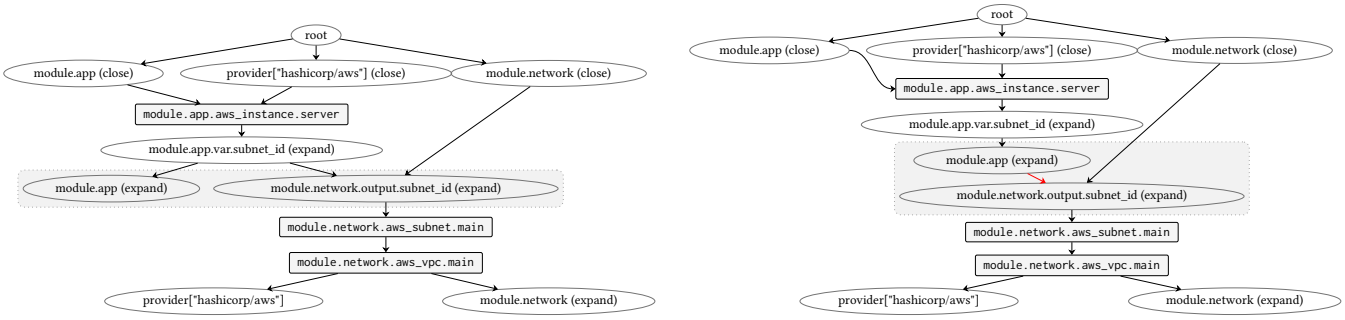
Expression references used on module inputs require special attention. The dependency they create targets the input variable and not the module itself. For this reason, a reader may mistakenly infer a dependency from the module to the dependee. We call these non-existing dependencies *spurious dependencies*. We further define a *possibly redundant dependency* as an explicit dependency that is implied by other dependencies in the configuration, including spurious dependencies.

In Listing 1, the `app` module receives its `subnet_id` from the `network` module. The expression `module.network.subnet_id` already allows Terraform to infer the required dependency: the input variable `subnet_id` of `app` depends on the corresponding output of `network`. The highlighted `depends_on` declaration is stronger than the inferred dependency, since the whole `app` module will now depend on the output of the `network` module. This may reflect the intended behavior for reasons not captured in the Terraform code, but it may also be indicative of a mistake. This dependency is an example of a possibly redundant dependency.

Listing 1: Terraform configuration containing redundant and possibly redundant dependencies

```
// main.tf
module "network" {
  source = "./modules/network"
}
module "app" {
  source     = "./modules/app"
  subnet_id  = module.network.subnet_id
  depends_on = [module.network.subnet_id]
}
// modules/network/main.tf
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}
resource "aws_subnet" "main" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"
  depends_on = [aws_vpc.main]
}
output "subnet_id" {
  value = aws_subnet.main.id
}
// modules/app/main.tf
variable "subnet_id" {
  type = string
}
resource "aws_instance" "server" {
  ami           = "ami-123"
  instance_type = "t2.micro"
  subnet_id     = var.subnet_id
}
```

Possibly redundant dependencies can affect how Terraform plans infrastructure changes, since these may convey less precise information about the values that create the dependency and may consequently lead to more conservative plans. For this reason, the Terraform documentation [11] recommends using `depends_on` only as a last resort. Figure 1a shows the dependency graph for Listing 1 without redundant dependencies. Figure 1b shows the dependency graph with the possibly redundant dependency. Since there is now an explicit dependency from the `app` module on `module.network.subnet_id`, all of the other components of the



(a) Without explicit dependencies.

(b) With a possibly redundant dependency (in red).

Figure 1: Terraform dependency graphs generated by the `terraform graph` utility for Listing 1. Terraform introduces vertices not only for resources, but also for providers, input and output variables, as well as dedicated *expand* and *close* vertices for modules. Every resource depends on its module's expand vertex, and the close vertex depends on every resource within the module. Boxes correspond to Terraform resources, while ovals correspond to other types of Terraform nodes.

app module become dependent as well. This results in a graph that is less parallelizable.

Formally, let d_e denote *explicit dependency* and let $d_e^{(i)}(A, B)$ denote the i-th explicit dependency from object A to object B introduced by `depends_on`. Let D_r denote the dependencies inferred from expression references, D_e the set of explicit dependencies, and D_s the set of spurious dependencies. We classify $d_e^{(i)}(A, B)$ as *redundant* when the dependency it imposes follows from the transitive closure of $(D_r \cup D_e) \setminus \{d_e^{(i)}(A, B)\}$, and *possibly redundant* if it follows from the transitive closure of $(D_r \cup D_e \cup D_s) \setminus \{d_e^{(i)}(A, B)\}$.

2.2 Detecting Redundant Dependencies

We abstract a configuration as a Directed Acyclic Graph (DAG) whose vertices represent Terraform resources, providers, input and output variables, local nodes, and data nodes, and whose edges represent *implicit*, *explicit*, or *spurious* dependencies.

The algorithm used by `depends_off` is shown in Algorithm 1. Starting with the full dependency graph, it considers one explicit edge at a time. That edge is removed from the graph, and the algorithm checks if there is a path between the edge's endpoints using only implicit and explicit edges. If a path exists, the edge corresponds to a redundant dependency.

Edges that are redundant are not added back to the graph. If they were, the algorithm would over-report in some cases. For instance, when the graph consists of vertices A and B, and there are two explicit edges from A to B, both edges would be reported as redundant, but only one of them is.

When there is no path between the dependency's endpoints, the algorithm tries to find a path using not only implicit and explicit edges but also spurious edges. If such a path exists, the dependency is a possibly redundant dependency. In either case, the edge is added to the graph, since it is not redundant and may be needed for other paths.

Implementation. We implement our approach in Python, in a tool named `depends_off`. We use `glitch-python-hcl2` [13], an adapted version of the `hcl2` Python package [1]. The tool additionally produces dependency graphs and SARIF reports, facilitating both inspection of its findings and integration with existing development workflows. `depends_off` is open-source and available at: https://github.com/infragov-project/depends_off

Algorithm 1 Detection of redundant dependencies

```
1:  function DetectRedundant(G)
2:      Explicit ← ExplicitEdges(G)
3:      Redundant, PossiblyRedundant ← [ ], [ ]

4:      for all Edge ∈ Explicit do
5:          (A, B) ← Vertices(Edge)
6:          Remove(G, Edge)

7:          if Path(G, A, B) then
8:              Add(Redundant, Edge)
9:          else
10:             if PathWithSpurious(G, A, B) then
11:                 Add(PossiblyRedundant, Edge)
12:             end if
13:             Add(G, Edge)
14:         end if
15:     end for

16:     return Redundant, PossiblyRedundant
17: end function
```

3 Empirical Study

In this section, we describe the research questions that guide our empirical study, the dataset used, and the methodology we followed.

3.1 Research Questions

We consider the following research questions:

- **RQ1.** How prevalent are redundant dependencies in Terraform repositories?
- **RQ2.** What characteristics are associated with redundant dependencies?

- **RQ3.** Which repository characteristics are associated with dependency redundancy?
- **RQ4.** What is the impact of removing redundant dependencies?

3.2 Dataset

We use *TerraDS* [5] version 2 [6], a large-scale dataset of publicly available Terraform programs collected from GitHub. TerraDS includes repositories containing HashiCorp Configuration Language (HCL) files that have permissive open-source licenses and valid Terraform modules, while excluding forks, repositories with non-permissive licenses, repositories that could not be cloned, and repositories without valid modules. The resulting dataset comprises 82,749 repositories created up to January 2026.

Overall, TerraDS comprises 394,004 Terraform modules and 2,497,629 resources. A module corresponds to a directory containing at least one Terraform `.tf` file. The dataset provides both the original HCL source code and structured metadata, including repository information, module paths, referenced providers and module calls, and resource names, types, and providers.

3.3 Methodology

Our analysis consists of three main stages: repository extraction, module discovery, and dependency analysis. We first extracted the source-code archive of each repository in TerraDS. Five repositories could not be successfully extracted and were therefore excluded from the analysis. We then traversed the remaining repositories to identify Terraform modules, resulting in 246,676 candidate modules. Each candidate module was analyzed using `depends_off`. Before detecting redundant dependencies, the tool parses the module and constructs its dependency graph. Of the 246,676 candidate modules, 19,929 could not be parsed and were excluded, mainly due to the use of modern Terraform features unsupported by the parser version used in our analysis. An additional 85 modules failed during the subsequent analysis stage. After these exclusions, `depends_off` successfully analyzed 226,662 Terraform modules.

To study the impact of redundant dependencies, we measured changes in dependency graphs after removing each possibly redundant dependency. We focused on these dependencies because they may impose additional ordering constraints on the graph and can therefore affect its available parallelism. For each case, we fetched the corresponding repository, ran `terraform init` and `terraform graph` to obtain the original graph, removed the diagnosed explicit dependency, and ran `terraform graph` again to obtain the modified graph.

Out of the 1,063 diagnostics of possibly redundant dependencies, we were able to measure the associated dependency graphs for 829. For the remaining 234, the corresponding repositories failed during `terraform init`. These failures included, among others, module instantiations with incorrect arguments, missing local module source code, invalid or missing function arguments, and missing required provider configurations.

4 Results

In this section, we present and discuss the results of our empirical study, organized around the research questions in Section 3.1.

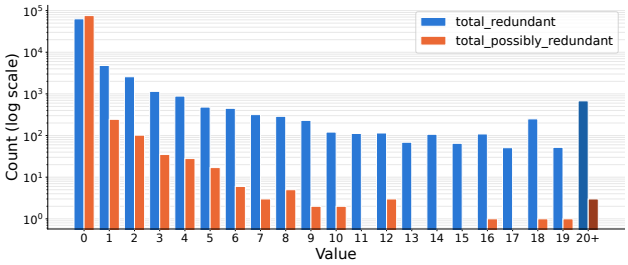


Figure 2: Distribution of number of detected problems per repository (values 0-19, tail bucketed as 20+).

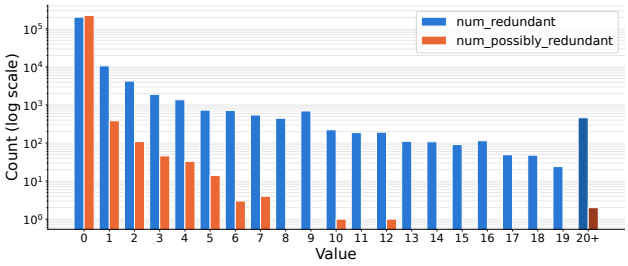


Figure 3: Distribution of number of detected problems per module (values 0–19, tail bucketed as 20+).

4.1 RQ1: Prevalence

We first investigate the prevalence of redundant dependencies in our dataset, considering both repositories and individual Terraform modules. Across the 76,296 repositories and 226,662 modules analyzed, `depends_off` detected a total of 89,570 redundant dependencies and 1,063 possibly redundant dependencies. Thus, redundant dependencies are substantially more prevalent than cases for which redundancy cannot be established conclusively.

Normalized by the size of the dataset, this corresponds to an average of 0.40 redundant dependencies and 0.005 possibly redundant dependencies per module. At the repository level, repositories contain an average of 1.17 redundant and 0.014 possibly redundant dependencies. In terms of prevalence, 17% of repositories contain at least one redundant dependency, compared with only 0.6% containing at least one possibly redundant dependency.

Figure 2 shows the distribution of the number of detected dependencies per repository. The distribution is highly skewed: most repositories contain no redundant dependencies, while among affected repositories, small numbers of redundant dependencies are the most common. Nevertheless, the long tail indicates that some repositories contain substantially larger numbers of redundant dependencies, with several hundred repositories containing 20 or more. This suggests that, although redundancy is absent from the majority of repositories, it can occur repeatedly within repositories where it is present.

A similar pattern emerges at the module level. Approximately 10% of the analyzed modules contain at least one redundant dependency, while only 0.3% contain at least one possibly redundant dependency. As shown in Figure 3, the distribution is again strongly

Table 1: Goodness-of-fit tests comparing the diagnostic occurrence distribution against the TerraDS corpus baseline (for Dependee $n = 61313$, for Depended $n = 48172$).

Side	Variable	Categories	χ^2	p-value	Cohen's w
Dependee	Type	1213	297,964.67	< 0.001	2.20
Dependee	Provider	180	72,876.55	< 0.001	1.09
Depended	Type	1002	104,297.42	< 0.001	1.47
Depended	Provider	163	11,414.92	< 0.001	0.49

Table 2: Top 10 resource types by percentage of instances that appear in redundant dependencies as dependee, together with the corresponding absolute number of instances. Resource types with a total number of instances below the median were excluded.

Resource Type	%	Total
`aci_ranges`	99.67%	2,705
`aws_network_interface_sg_attachment`	84.72%	1,697
`kubernetes_config_map_v1_data`	43.41%	369
`aws_ec2_transit_gateway_route_table`	29.28%	200
`aws_ec2_transit_gateway_vpc_attachment`	28.92%	317
`oci_core_vnic_attachment`	26.20%	137
`azurerm_cosmosdb_account`	22.90%	177
`aws_codepipeline`	22.44%	281
`azurerm_network_interface_security_group_association`	19.52%	918
`google_cloudfunctions2_function`	19.07%	82

skewed toward modules with no detected dependencies, followed by modules containing only a small number. However, a long tail is also visible at this level, including hundreds of modules with 20 or more redundant dependencies. The presence of this tail shows that redundant dependencies are not necessarily isolated occurrences within affected modules.

The repository- and module-level results also provide complementary views of prevalence. While approximately one in ten modules is affected, approximately one in six repositories contains at least one redundant dependency. This difference is expected because a repository may contain multiple Terraform modules and is considered affected whenever redundancy occurs in any of them. Overall, the results indicate that redundant dependencies are not ubiquitous, but are sufficiently widespread to constitute a recurring phenomenon in real-world Terraform code.

RQ1: How prevalent are redundant dependencies in Terraform repositories? Across 76,296 repositories and 226,662 Terraform modules, we detected 89,570 redundant dependencies. Redundancy occurs in approximately **17% of repositories** and **10% of modules**. Although most repositories and modules contain no redundant dependencies, affected projects can contain many instances, resulting in a pronounced long-tailed distribution.

Table 3: Top 10 providers by percentage of instances that appear in redundant dependencies as dependee, together with the corresponding absolute number of instances. Providers with a total number of instances below the median were excluded.

Resource Provider	%	Total
`aci`	31.34%	3,669
`turbot`	21.01%	237
`null`	11.94%	4,760
`rancher2`	11.78%	72
`helm`	10.93%	1,851
`camc`	9.38%	87
`volterra`	8.34%	109
`ciphertrust`	7.43%	60
`confluent`	6.92%	113
`snowflake`	6.35%	114

4.2 RQ2: Characteristics

We investigate whether redundant dependencies exhibit recurring patterns with respect to the Terraform objects they connect. Specifically, we analyze the *resource type* and *provider* of the resources participating in redundant dependencies. We extract the resource type from each resource reference and derive its provider from the corresponding type prefix. For example, Listing 1 declares resources of types `aws_vpc`, `aws_subnet`, and `aws_instance`, which we associate with the `aws` provider based on their `aws_` prefix. Some resources are instead intrinsic to Terraform and have no provider, most notably `null_resource`, which is commonly used to integrate imperative code into declarative Terraform configurations. Since explicit dependencies may also reference constructs such as modules, for which these characteristics are not directly applicable, this analysis is restricted to resources.

First, we compare the observed distributions in redundant dependencies against their corresponding distributions in the TerraDS corpus. We consider the dependee and depended sides separately and use chi-squared goodness-of-fit tests for both resource type and provider, grouping low-frequency categories into an *Other* category to satisfy the expected-frequency requirements of the test. Table 1 presents the results.

All four distributions differ significantly from the TerraDS baseline ($p < 0.001$). For resource types, the differences are particularly pronounced, with large effect sizes for both the dependee ($w = 2.20$) and depended ($w = 1.47$) sides. Provider also exhibits a large effect for the dependee ($w = 1.09$), but a smaller effect for the depended side ($w = 0.49$). These effect sizes indicate that the observed differences are not merely a consequence of the large sample size. Due to space constraints, we focus on the dependee side for the remainder of this section, as it exhibits the largest effect sizes for both providers and resource types.

Table 2 reports the top ten resource types by percentage of instances being the dependee in redundant dependencies. The list is led by `aci_ranges`, for which the detected 2705 resources correspond to 99.67% of that type's instances. Following it, we have `aws_network_interface_sg_attachment`, which has 84.72% of its instances as part of a redundant dependency. After observing

the diagnosed code for specifically these two resource types, we observe that the extremely high percentage is mainly due to heavy duplication of problematic code in a single or a small number of repositories. Following these two resource types, we still have four different resource types having around the 30-40% of their instances present in redundant dependencies, followed by other four resource types that still maintain a percentage around 20%. These percentages indicate that these resource types are overrepresented among redundant dependencies relative to their prevalence in the overall corpus.

At the provider level, we don't have specific cases with percentages close to 100%. As shown in Table 3, `aci` leads the providers by having 31.34% of its resources being the dependee of a redundant dependency. Another interesting example is the `null` provider, being the provider for which the `null_resource` type belongs, having a significant percentage and a high absolute number of redundant dependencies. We believe this is intrinsically related with the purpose of `null_resource`, leading to developers overusing explicit dependencies in these resources. `helm` seems to be another example of a provider with a significant percentage of resources being part of a redundant dependency, while still having a high total number of resources. The remaining providers in the top 10 were less used providers that, while having a significant percentage of resources present in redundant dependencies, had a smaller number of cases. While we decided to ignore resource types and providers that had a total number of instances in the dataset below the median, this still leads us to believe that we are dealing with more niche providers, for which a small number of cases might lead to a significant percentage.

Overall, redundant dependencies are *not uniformly distributed across Terraform constructs*. Resource type is strongly associated with redundancy on both sides of a dependency, while provider is particularly informative for the dependee. This suggests that characteristics of the Terraform objects involved in a dependency can help identify where redundancy is more likely to occur.

> **RQ2: What characteristics are associated with redundant dependencies?** Redundant dependencies exhibit **strong resource- and provider-specific patterns**. Resource-type distributions differ substantially from the TerraDS baseline on both the dependee and depended sides, with large effect sizes (Cohen's $w = 2.20$ and $w = 1.47$, respectively). Provider differences are also pronounced for the dependee ($w = 1.09$), but smaller for the depended side ($w = 0.49$). Overall, redundant dependencies are disproportionately associated with particular Terraform constructs, especially on the dependee side.

4.3 RQ3: Repositories

We investigate whether dependency redundancy is associated with characteristics of the repositories in which it occurs. To account for differences in the number of dependencies across repositories, we define the *redundant dependency rate* as the number of redundant dependencies divided by the total number of explicit dependencies in a repository. Since this metric is undefined for repositories without explicit dependencies, this analysis considers the 28,710 repositories containing at least one explicit dependency (around 38% of the total dataset).

Table 4: Linear regression of redundant dependency rate on repository-level variables ($n = 28710$).

Variable	Slope	r	r^2	p-value
Repo age (days)	1.5088×10^{-5}	0.0355	0.0013	1.745×10^{-9}
Repo activity (days)	2.3528×10^{-5}	0.0512	0.0026	3.733×10^{-18}
Lines of code	2.0282×10^{-7}	0.0044	0.0000	0.4512
Number of modules	4.1141×10^{-5}	0.0036	0.0000	0.5407

We first examine four repository-level characteristics: repository age, repository activity, lines of code (LoC), and number of Terraform modules. Repository age and activity are measured in days relative to the TerraDS cutoff date, with activity capturing the time since the repository's last commit. These two variables are obtained from the TerraDS metadata, while LoC and the number of modules are computed during our analysis.

Table 4 reports the results of fitting a linear regression between each characteristic and the redundant dependency rate. Overall, none of the four characteristics exhibits a practically meaningful linear association with the redundant dependency rate. Repository age ($r = 0.0355$, $r^2 = 0.0013$) and repository activity ($r = 0.0512$, $r^2 = 0.0026$) exhibit statistically significant positive associations ($p < 0.001$). However, their effect sizes are negligible: even the strongest association, repository activity, explains only 0.26% of the variance in the redundant dependency rate. Given the large sample size ($n = 28{,}710$), these small effects can reach statistical significance despite having little practical explanatory power.

For LoC ($r = 0.0044$, $p = 0.4512$) and number of modules ($r = 0.0036$, $p = 0.5407$), we find neither statistically significant nor practically meaningful linear associations.

Taken together, these results suggest that dependency redundancy is not readily explained by the repository-level characteristics considered in our analysis. In particular, larger repositories, whether measured by LoC or number of modules, do not exhibit higher redundant dependency rates, and neither repository age, activity, nor archival status provides meaningful explanatory power.

> **RQ3: Which repository characteristics are associated with dependency redundancy?** We find *no practically meaningful association* between redundant dependency rate and repository age, activity, size (LoC), number of modules, or archival status. This suggests that dependency redundancy is not primarily a property of larger, older, or less actively maintained repositories, but is instead likely associated with characteristics of the Terraform configurations themselves.

4.4 RQ4: Impact

To evaluate the impact of redundant dependencies, we compared the dependency graph produced by the Terraform CLI before and after removing a single diagnosed explicit dependency. We focused on dependencies classified as possibly redundant, as these dependencies impose additional ordering constraints on the graph and can therefore directly limit the available parallelism.

Table 5: Impact of removing possibly redundant dependencies on Terraform dependency graphs. Values report the relative reduction in critical path length and the relative increase in mean parallelism with respect to the original graph. Results are reported only for cases where the corresponding metric improved ($n = 71$).

Metric	Mean	Median	I.Q.R.	P_{90}	Max
Critical Path Length	17.2%	14.3%	[11.1%, 22.2%]	28.6%	50%
Mean Parallelism	18.2%	14.3%	[11.1%, 25.0%]	33.3%	57.1%

The primary graph-level metric was the critical path length (CPL), defined as the length of the longest path in the resulting directed acyclic graph. We additionally considered mean parallelism, defined as the number of nodes divided by the critical path length plus one, i.e., (#nodes/(CPL + 1)). Since removing a dependency does not alter the number of nodes, an increase in mean parallelism can only result from a reduction in critical path length.

Out of the 829 graphs for which the impact of dependency removal was successfully measured, 71 exhibited a reduction in critical path length and, consequently, an increase in mean parallelism. Thus, 8.6% of the analyzed graphs were affected by the removal of at least one diagnosed possibly redundant dependency (see Table 5 for a detailed breakdown of metric changes on the affected graphs).

For the affected graphs, the reduction in critical path length had a median relative reduction of 14.3%, with an interquartile range (IQR) of 11.1%–22.2%. The mean improvement was 17.2%, while the 90th percentile reached 28.6%, with a maximum reduction of 50.0%.

The improvements in mean parallelism were slightly larger. The median increase was 14.3%, with an IQR of 11.1%–25.0%. The arithmetic mean increase was 18.2%, while the geometric mean of the improvement ratio was 17.7%. The 90th percentile reached 33.3%, and the maximum observed increase was 57.1%.

> **RQ4: What is the impact of removing redundant dependencies?** Removing possibly redundant dependencies affected the dependency graph in 8.6% of the analyzed cases. Among affected graphs, the median reduction in critical-path length and increase in mean parallelism were both 14.3%, showing that unnecessary explicit dependencies can meaningfully constrain Terraform's available parallelism.

5 Discussion

In this section, we discuss the practical implications of our findings and provide recommendations for practitioners, before presenting the threats to the validity of our study.

5.1 Practical Implications

Our findings have several implications for practitioners developing and maintaining Terraform configurations.

Review existing explicit dependencies. The long-tailed distribution observed in RQ1 shows that redundant dependencies can accumulate within individual projects. We therefore recommend periodically reviewing existing `depends_on` declarations, for example by integrating `depends_off` into code review or CI workflows. In more conservative scenarios, organizations can automatically enforce policies that prevent redundant explicit dependencies via Policy as Code (PaC) rules [15, 20]. Prior work also suggests that potentially problematic IaC constructs can be interpreted differently by reviewers, so ambiguous cases may benefit from confirmation by multiple developers [14].

Prioritize diagnostics based on the Terraform constructs involved. Redundancy is not uniformly distributed across resource types and providers: their distributions among redundant dependencies differ substantially from their prevalence in the underlying corpus. Dependency reviews can therefore prioritize constructs frequently associated with redundancy. In particular, the frequent occurrence of `null_resource` suggests that explicit dependencies involving this resource warrant additional scrutiny.

Use repository characteristics as proxies for risk with caution. We found no practically meaningful association between redundancy and repository age, activity, size, number of modules, or archival status. Dependency analysis should therefore not be restricted to large, old, or less actively maintained repositories, but instead applied directly to Terraform configurations across projects.

Give possibly redundant dependencies additional attention. Possibly redundant dependencies may encode intentional constraints not captured by the configuration and should therefore not be removed automatically. However, they can also unnecessarily constrain Terraform's dependency graph. Removing one such dependency reduced the critical path in 8.6% of the cases we could measure; among affected graphs, the median reduction in critical-path length and increase in mean parallelism were both 14.3%. We therefore recommend reporting these cases for manual inspection.

Integrate dependency analysis into Terraform quality assurance. Overall, dependency analysis can complement existing Terraform quality checks. A practical workflow is to automatically flag dependencies established as redundant while reporting possibly redundant dependencies for developer inspection. This allows unnecessary constraints to be identified early while leaving semantically ambiguous cases to developers.

5.2 Threats to Validity

Internal validity. Our results depend on the accuracy of our tool, `depends_off`, and its reconstruction of Terraform dependencies. In particular, limitations of `glitch-python-hcl2` may cause some expression references, and thus some redundant dependencies, to be missed. Moreover, modules that could not be parsed or analyzed were excluded from the study.

External validity. Our study is based on public, permissively licensed GitHub repositories from TerraDS. Although the dataset provides broad coverage of Terraform code, our findings may not generalize to private or industrial repositories or to future versions of Terraform.

Impact analysis. RQ4 measures changes to dependency-graph structure rather than actual provisioning time, so increased parallelism may not translate directly into faster execution. Moreover, the analysis covers only the subset of diagnosed cases for which `terraform init` and graph construction succeeded, which may introduce selection bias.

6 Related Work

In this section, we review prior work on the analysis of IaC, with particular attention to approaches targeting Terraform configurations and their relationship to our dependency analysis.

Infrastructure-as-Code Analysis Prior work has extensively studied the automated analysis of IaC, with particular emphasis on defects, code smells, and security issues. Rahman et al. [19] identified seven recurring security smells in IaC scripts and proposed SLIC, a rule-based detector for these smells. Subsequent work refined these rules based on practitioner feedback [21] and also investigated more sophisticated program analyses, including control-flow and data-flow analysis for detecting security smells in Ansible [16]. Other approaches have used machine learning to detect defects and anti-patterns in IaC configurations [3, 4, 9, 17].

Several tools support static analysis across multiple IaC technologies. Checkov [18] provides a common framework for checking different IaC languages, whereas KICS [8] and GLITCH [22, 24] represent configurations using language-independent intermediate representations, enabling analyses to be reused across technologies. These approaches do not analyze the dependency graph induced by IaC configurations. In contrast, our work analyzes the Terraform dependency graph to determine whether explicit dependencies duplicate dependencies already induced by expression references.

Dependency analysis has received comparatively less attention. Sotiropoulos et al. [26] detect missing dependencies in Puppet by dynamically observing system calls, while Shambaugh et al. [25] statically model the effects of Puppet manifests and use automated reasoning to verify properties, including determinism. These works demonstrate the importance of correctly specifying dependencies in IaC, but target Puppet's substantially different execution model rather than Terraform.

Terraform Analysis Terraform-specific analysis has predominantly focused on detecting misconfigurations and security problems. Tools such as tfsec [2] provide rule-based checks for Terraform configurations, while Hu et al. [12] empirically studied static-analysis alerts produced for Terraform manifests. More recently, Vo et al. [28] investigated LLMs for detecting code smells in Terraform, and TerraFault [29] uses dependency graphs together with LLM agents to discover bugs introduced by infrastructure updates. TerraFault is particularly related in its use of Terraform dependency graphs, but targets bugs arising from configuration updates rather than redundant dependency declarations.

To the best of our knowledge, prior work has not systematically studied *redundant dependencies* in Terraform. Our work fills this gap by introducing a static analysis for detecting explicit `depends_on` relationships that are already implied by Terraform references, and by conducting a large-scale empirical study to characterize their prevalence and properties in real-world Terraform configurations.

7 Conclusion

Our study shows that redundant dependencies are a recurring and consequential phenomenon in Terraform: they affect 17% of repositories and 10% of modules, and are strongly associated with particular Terraform constructs. Moreover, removing possibly redundant dependencies can meaningfully improve dependency graphs, with affected cases exhibiting a median 14.3% reduction in critical-path length. By detecting these dependencies, `depends_off` enables developers to identify unnecessary constraints before they impact infrastructure provisioning. Our findings motivate dependency analysis as a first-class component of Terraform quality assurance.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 (DOI: 10.54499/UID/50021/2025), UID/PRR/50021/2025 (DOI: UID/PRR/50021/2025), and co-funded by the European Union through the Lisboa 2030 Programme (ERDF) and by national funds through FCT, I.P., under project no. 15018 (project SafeIaC, DOI: 10.54499/2023.18089.ICDT). L. Barbosa was funded by the 'InfraGov' project, with ref. n. 2024.07411.IACDC (DOI: 10.54499/ 2024.07411.IACDC).

References

[1] Amplify Education. 2026. *python-hcl2.* https://pypi.org/project/python-hcl2/
[2] Aqua Security. 2026. *tfsec.* https://github.com/aquasecurity/tfsec
[3] Mahi Begoug, Moataz Chouchen, Ali Ouni, Eman Abdullah Alomar, and Mohamed Wiem Mkaouer. 2024. Fine-grained just-in-time defect prediction at the block level in Infrastructure-as-Code (IaC). In *Proceedings of the 21st International Conference on Mining Software Repositories.* 100–112.
[4] Nemania Borovits, Indika Kumara, Parvathy Krishnan, Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, Damian A Tamburri, and Willem-Jan van den Heuvel. 2020. DeepIaC: deep learning-based linguistic anti-pattern detection in IaC. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation.* 7–12.
[5] Christoph Bühler, David Spielmann, Roland Meier, and Guido Salvaneschi. 2025. Terrads: A dataset for terraform hcl programs. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR).* IEEE, 654–658.
[6] Christoph Bühler, David Spielmann, Roland Meier, and Guido Salvaneschi. 2026. *TerraDS: A Dataset for Terraform HCL Programs.* doi:10.5281/zenodo.20339474
[7] Carolina Carreira, Nuno Saavedra, Alexandra Mendes, and João F Ferreira. 2025. The Ultimate Configuration Management Tool? Lessons from a Mixed Methods Study of Ansible's Challenges. In *Proceedings of the 30th International Conference on Evaluation and Assessment in Software Engineering (EASE).*
[8] Checkmarx. 2026. *KICS - Keeping Infrastructure as Code Secure.* https://www.kics.io
[9] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian A Tamburri. 2021. Within-project defect prediction of infrastructure-as-code using product and process metrics. *IEEE Transactions on Software Engineering* 48, 6 (2021).
[10] HashiCorp. [n. d.]. *Terraform Documentation: Dependency Graph.* Accessed: 2026-01-14. https://developer.hashicorp.com/terraform/internals/graph
[11] HashiCorp. 2026. *depends_on reference.* Accessed: 2026-08-12. https://developer.hashicorp.com/terraform/language/meta-arguments/depends_on#processing-and-planning-consequences
[12] Hanyang Hu, Yani Bu, Kristen Wong, Gaurav Sood, Karen Smiley, and Akond Rahman. 2023. Characterizing static analysis alerts for terraform manifests: An experience report. In *2023 IEEE Secure Development Conference (SecDev).* IEEE.
[13] João Gonçalves. 2024. *glitch-python-hcl2.* https://pypi.org/project/glitch-python-hcl2/
[14] Sogol Masoumzadeh, Nuno Saavedra, Rungroj Maipradit, Lili Wei, João F Ferreira, Dániel Varró, and Shane McIntosh. 2025. Do Experts Agree About Smelly Infrastructure? *IEEE Transactions on Software Engineering* 51, 5 (2025), 1472–1486.
[15] Ruben Opdebeeck, Mahmoud Alfadel, Akond Rahman, Yutaro Kashiwa, João F Ferreira, Raula Gaikovina Kula, and Coen De Roover. 2026. An empirical study of policy as code: Adoption, purpose, and maintenance. In *Proceedings of the 23rd International Conference on Mining Software Repositories.* 249–261.
[16] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2023. Control and data flow in security smell detection for infrastructure as code: Is it worth the effort?. In *IEEE/ACM 20th Int. Conf. on Mining Software Repositories (MSR).* IEEE.
[17] Stefano Dalla Palma, Majid Mohammadi, Dario Di Nucci, and Damian A Tamburri. 2020. Singling the odd ones out: a novelty detection approach to find defects in infrastructure-as-code. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation.*
[18] Prisma Cloud. 2026. *Checkov.* https://www.checkov.io/
[19] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE, 164–175.

[20] Jimmy Ray. 2024. *Policy as Code* (1st ed.). O'Reilly Media.
[21] Sofia Reis, Rui Abreu, Marcelo d'Amorim, and Daniel Fortunato. 2022. Leveraging practitioners' feedback to improve a security linter. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[22] Nuno Saavedra and João F Ferreira. 2022. Glitch: Automated polyglot security smell detection in infrastructure as code. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*. 1–12.
[23] Nuno Saavedra, João F Ferreira, and Alexandra Mendes. 2025. InfraFix: Technology-Agnostic Repair of Infrastructure as Code. In *34th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 41–45.
[24] Nuno Saavedra, João Gonçalves, Miguel Henriques, João F Ferreira, and Alexandra Mendes. 2023. Polyglot code smell detection for infrastructure as code with GLITCH. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2042–2045.
[25] Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 416–430.
[26] Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. Practical fault detection in puppet programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 26–37.
[27] Stack Overflow. 2025. Stack Overflow Developer Survey 2025. [Accessed 29-06-2026]. https://survey.stackoverflow.co/2025
[28] Quoc-Huy Vo, Ha Dao, and Kensuke Fukuda. 2025. Harnessing the Power of LLMs for Code Smell Detection in Terraform Infrastructure as Code. In *IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE.
[29] Yiming Xiang, Zhenning Yang, Jingjia Peng, Hermann Bauer, Patrick Tser Jern Kon, Yiming Qiu, and Ang Chen. 2025. Automated bug discovery in cloud infrastructure-as-code updates with llm agents. In *2025 IEEE/ACM International Workshop on Cloud Intelligence & AIOps (AIOps)*. IEEE, 20–25.