Fine-Grained Just-In-Time Defect Prediction at the Block Level in Infrastructure-as-Code (IaC)

Mahi Begoug
ETS Montreal, University of Quebec
mahi.begoug.1@ens.etsmtl.ca

Moataz Chouchen
ETS Montreal, University of Quebec
moataz.chouchen.1@ens.etsmtl.ca

Ali Ouni
ETS Montreal, University of Quebec
ali.ouni@etsmtl.ca

Eman Abdullah AlOmar
Stevens Institute of Technology
ealomar@stevens.edu

Mohamed Wiem Mkaouer
University of Michigan-Flint
mmkaouer@umich.edu

ABSTRACT

Infrastructure-as-Code (IaC) is an emerging software engineering practice that leverages source code to facilitate automated configuration of software systems' infrastructure. IaC files are typically complex, containing hundreds of lines of code and dependencies, making them prone to defects, which can result in breaking online services at scale. To help developers early identify and fix IaC defects, research efforts have introduced IaC defect prediction models at the file level. However, the granularity of the proposed approaches remains coarse-grained, requiring developers to inspect hundreds of lines of code in a file, while only a small fragment of code is defective. To alleviate this issue, we introduce a machine-learning-based approach to predict IaC defects at a fine-grained level, focusing on IaC blocks, i.e., small code units that encapsulate specific behaviours within an IaC file. We trained various machine learning algorithms based on a mixture of code, process, and change-level metrics. We evaluated our approach on 19 open-source projects that use Terraform, a widely used IaC tool. The results indicated that there is no single algorithm that consistently outperforms others in 19 projects. Overall of the six algorithms, we observed that the LightGBM model achieved a higher average of 0.21 in terms of MCC and 0.71 in terms of AUC. Models analysis reveals that the developer's experience and the relative number of added lines tend to be the most important features. Additionally, we found that blocks belonging to the most frequent types are more prone to defects. Our defect prediction models have also shown sensitivity to concept drift, indicating that IaC practitioners should regularly retrain their models.

KEYWORDS

Defect Prediction, Infrastructure-as-Code, IaC, Terraform

ACM Reference Format:
Mahi Begoug, Moataz Chouchen, Ali Ouni, Eman Abdullah AlOmar, and Mohamed Wiem Mkaouer. 2018. Fine-Grained Just-In-Time Defect Prediction at the Block Level in Infrastructure-as-Code (IaC). In *Proceedings of 21st International Conference on Mining Software Repositories (MSR 2024).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

MSR 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

A recent trend in the DevOps movement's practice attempts to automate the configuration process of infrastructure deployment, known as *Infrastructure-as-Code* (IaC). IaC is an emerging software engineering practice that uses machine-readable files to provision and configure software infrastructures, diverging from conventional manual hardware setup or interactive configuration tools [54, 79, 88]. In IaC, practitioners can incorporate coding principles such as automated testing and version control. To improve the readability of IaC files, practitioners use *blocks* similar to methods or functions in traditional software development. These blocks are designed to describe behaviours for declaring infrastructure components. Given these significant advantages, various IaC tools, including Ansible [49], Terraform [35], and CloudFormation [42] have gained widespread adoption among practitioners [30]. To take advantage of its usage, several companies have recently integrated IaC into their development pipelines to streamline the implementation process of infrastructure changes, such as Shopify [25] and Uber [14].

Although IaC strives to automate the delivery process, various challenges can hinder its adoption in practice, including technical debt and bugs within IaC files [78]. It is also challenging to deal with the complexity of IaC code related to managing different components of the configuration infrastructure. Therefore, the IaC code could be susceptible to misconfigurations and defects, which could have severe consequences [74]. In practice, IaC defects are prevalent and may affect millions of end-users causing service outages, and systems unavailability [38]. A recent example manifests in the `terraform-aws-modules/terraform-aws-eks` [1] module which is widely used by IaC practitioners to orchestrate and manage easily Kubernetes clusters in AWS environments using Terraform files. However, the module users have faced a significant issue since 2020 [44]. This issue revolves around cyclic dependency errors between infrastructure components, and several users reported that this problem persisted and caused infrastructure outages [29].

Early studies have explored the nature of IaC defects by providing metrics to assess the quality of IaC files. For example, Rahman et al. [78] proposed eight patterns to categorize a defect in IaC files. Dalla et al. [38] and Rahman et al. [82] used machine learning (ML) techniques to build defect prediction models based on IaC source code metrics and historical change. Defect prediction (DP)

models are a strategic allocation of testing and inspection resources, focusing on software artifacts that are more prone to defects. Furthermore, Dalla et al. [38] constructed the IaC defect models by extracting metrics based on the releases of Ansible-based projects. However, two limitations can hinder the adoption of the proposed IaC defect models. First, since an IaC file can be divided into blocks that abstract the infrastructure configuration, not all blocks within a defective file contain defects. Consequently, IaC practitioners can still waste time and effort inspecting clean blocks that may not contain defects. Therefore, IaC defect models are still needed at more fine-grained granularity. Secondly, relying on defect prediction at the release level makes it difficult to pinpoint which practitioners should inspect the predicted IaC file, as numerous practitioners can frequently collaborate on the same file. Thus, there is a need for instantaneous prediction that allows direct inspection based on Just-In-Time defect prediction (JIT) models. JIT is a kind of defect prediction, known as change level, which predicts code changes in committed files that may be more prone to defects [43, 60, 68, 76]. JIT permits timely detection of potential commits when a commit is made, allowing for the avoidance of infrastructure server outages.

To address these issues, we introduce a fine-grained prediction approach to predict whether a block in an IaC file is likely to be defective. To provide IaC practitioners with early detection, our approach leverages Just-in-time (JIT) defect prediction. Our approach predicts the changed blocks that were altered by the added or deleted lines at the commit level. In our study, we investigate the within-project defect prediction in which each project has its defect prediction models. Our research is explicitly focused on Terraform-based projects, as Terraform is widely recognized and widely used for provisioning among IaC practitioners [2, 20]. To the best of our knowledge, no previous IaC study has investigated the defects prediction in Terraform-based projects. We conducted an empirical study to train and test six machine learning algorithms on 19 datasets collected from open-source Terraform-based projects. We train the models on the code, process, and change level metrics. Our study addresses the following research questions:

RQ1 (Model Performance): How effective is our approach in IaC defects prediction at the block level? RQ1 aims to determine a suitable model that could accurately predict defective blocks. The results show that ML algorithms can predict the defects at the block level in IaC files with a prominent performance. Meanwhile, our findings indicate that there is no single ML model that outperforms consistently better in all projects. We observed that LighGBM tend to be best best-performing model achieving an average score of 0.71 in AUC and 0.21 in MCC.

RQ2 (Feature Importance): Which features are the most influential for IaC defects prediction at the bloc level? RQ2 investigates the key features among code, process, and delta (*i.e.,* change) features. The top-3 features are the recent experience, the proportion of lines added within a changed block, and frequently changed block types.

RQ3 (Model Stability): To what extent does our approach address concept drift? RQ3 assesses the stability of our models over time as data can change, potentially leading to a degradation in the performance of prediction models due to shifts in the relationship between independent variables and the ground truth. Our results indicate that concept drift is common in the studied projects and training on more recent data improves the model's performance as compared to using older data in at least 13 out of 19 projects.

Replication Package. We provide a comprehensive replication package of our dataset and scripts available online [18].

Paper Organization. The remainder of this paper is divided as follows. In Section 2, we present the background and related works. In Section 3, we describe the study methodology, while in Section 5, we provide implications of our study, and we discuss the potential threats to our study in Section 6. Lastly, we draw our conclusions and future directions in Section 7.

2 BACKGROUND & RELATED WORKS

2.1 IaC & Terraform

Terraform is an infrastructure-as-code (IaC) ecosystem that automates infrastructure provisioning in various data centers and cloud providers [23]. It also supports multi-cloud provisioning, allowing users to work with different cloud resources such as AWS Lambda [12], Azure Active Directory [40], and Google Cloud Load Balancers [28]. The Terraform ecosystem consists of the following two key components: (i) *Terraform Core*, which serves as a compiler, executing configurations authored by Terraform practitioners and generating a file known as the *State File* that summarizes the dependencies between the various declared components; (ii) *Terraform Plugins*, which are modules that interpret and execute information within the State File and facilitate communication with cloud platforms through application programming interface [23]. Terraform offers numerous advantages to establishing in high-quality software system infrastructures.

Terraform allows practitioners to declaratively configure their manifests using the HashiCorp Configuration Language (HCL) [24]. In this ecosystem, HCL plays a pivotal role. HCL is a human-readable language used for composing IaC manifests, typically saved with the *".tf"* extension. Within HCL, developers encounter two fundamental elements: (i) arguments (*i.e.,* attributes), representing parameter key-value pairs, and (ii) blocks, which serve as containers for these attributes and nested blocks, as exemplified in Listing 1. Terraform categorizes these elements into eight block types: Resources, Data Sources, Providers, Variables, Outputs, Modules, Locals, and Terraform Configuration.

As an illustration of these blocks, Listing 2 presents an HCL configuration to create a virtual machine on the AWS platform. The code snippet first specifies the version of the AWS provider within the Terraform block and defines the version of the Terraform core. It then specifies the region of the data center where the infrastructure will be created. Subsequently, the `resource` block is used to describe the characteristics of the virtual machine, initializing attributes such as the `ami` (*i.e.,* identifier), defining the type of instance, and assigning a name. Additionally, `variable` and `locals` can be used to encapsulate certain characteristics, providing a way to refactor and avoid direct manipulation of resource attributes. Finally, the `output` block can be used to retrieve the public address of the virtual machine after its creation.

Listing 1: HCL Syntax

```
1 <BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
```

```
  <IDENTIFIER> = <EXPRESSION> # Argument
  <BLOCK LABEL> { # Nested Blocks
  }
}
```

Listing 2: HCL VM Creation Example

```
terraform {
  required_providers {
    aws = { source = "hashicorp/aws", version =
          "~> 4.16"
    }
  }
  required_version = ">= 1.2.0"
}

provider "aws" {
  region = "us-west-2"
}

locals {
  ami = "ami-830c94e3"
}

variable "instance_type" {
  description = "VM Instance Specification"
  default     = "t2.micro"
}

resource "aws_instance" "app_server" {
  ami           = local.ami
  instance_type = var.instance_type
  tags          = {
    Name = "ExampleAppServerInstance"
  }
}

output "instance_vm_name" {
  value = aws_instance.app_server.public_ip
}
```

2.2 Related Works

Recently, IaC has garnered increasing interest within the research community. Several studies align with our research exploring the adoption, challenges, and, notably, the defects associated with IaC.

Jiang et al. [54] studied the co-evolution between IaC and other code artifacts (build, test, production) to examine the growth in size and complexity of IaC files. They found that IaC scripts undergo recurring changes. Subsequently, Sharma et al. [88] explored code smells in IaC scripts and proposed 13 implementations and 11 design configuration smells related to Puppet projects. Rahman et al. [78] proposed a defect taxonomy for IaC scripts to help practitioners classify defective puppet files. To validate the proposed taxonomy, they surveyed 66 practitioners. They found that practitioners settled mainly on issues related to idempotency. Opdebeeck et al. [74] also investigated variables that could lead to infrastructure outages in the context of Ansible. They pinpointed six code smells related to Ansible's complex variable priority rules and lazy-evaluated template expressions. They observed an increasing trend in variable smells over time, emphasizing that it may take a considerable amount of time to fix and may also introduce new code smells. These results underscore the need for more extensive quality checks for the IaC code. Subsequently, Saavedra et al. [86] proposed a technology-agnostic framework called GLITCH, which identifies security smells in different IaC technologies. The approach identifies nine security smells in three IaC technologies (Ansible, Chef, or Puppet). GLITCH transforms the provided IaC code into an intermediate representation to streamline the tagging of security smells. When comparing their results with existing security detection tools, they found that GLITCH showed higher precision and recall compared to current state-of-the-art tools.

Concerning the prediction of defects in IaC, Rahman et al. [82] studied defective puppet files that could exist in open-source projects such as Miratis, Mozilla, OpenStack, and Wikimedia. First, they analyze the commit message to determine the defect-related commits. Then, they extracted the code properties that could characterize the changed Puppet file. They applied qualitative analysis to identify 12 source code properties correlating with defective Puppet files. They notice that *"lines of code"* and *"hard-coded string"* strongly correlate with defective IaC files. Subsequently, they constructed defect prediction models on different metrics such as code sources, process, implementation smells, and bag-of-words metrics and reported a precision of 0.70 ~ 0.78 and a recall of 0.54 ~ 0.67.

Recently, Dalla et al. [38] studied the defects that concern Ansible files. They proposed a complete machine learning pipeline for IaC defect predictions that collects Ansible files and their metrics, builds learners, and evaluates. To validate this pipeline, they analyzed 104 open-source projects on GitHub and extracted metrics at the release level. They employed five machine learning classifiers and found that RandomForest is the best-performing model. Moreover, they found that product metrics identify defective IaC scripts more accurately than process metrics.

While most previous IaC research has focused on defects and code smells to enhance configuration manifest quality, these studies have primarily concentrated on configuration management tools designed to aid developers in configuring and managing their applications after establishing infrastructures such as Puppet, Ansible, and Chef. These tools can be classified as Configuration-as-Code Tools. Our study complements those mentioned above to build a complete ontology for IaC. Our study studies defects in provisioning tools, focusing on Terraform files at the commit level (*i.e.,* change) using machine learning algorithms. To make it a more fine-grained technique, we focus on the changed blocks of configuration (*i.e.,* units) where the defects could occur, aiming to reduce the effort of inspecting large and complex Terraform files.

3 STUDY DESIGN

We aim to predict whether a modified Terraform block could be defective. This prediction helps developers with detailed viewpoints when analyzing intricate changes. To achieve this goal, the workflow of our study is depicted in Figure 1, which comprises three essential phases: (1) Context Selection, (2) Dataset Construction, (3) Model Construction, and (4) Statistical test Usage. In the following, we provide an overview of each of these phases.

3.1 Context Selection

Previous studies on Infrastructure as Code (IaC) focused mainly on configuration management tools [38, 78, 80]. Consequently, existing IaC datasets are limited to Chef, Puppet, and Ansible. To overcome this limitation, we use the GitHub Code Search API to pinpoint open-source projects that contain at least one Terraform file [15]. These Terraform files must have a *".tf"* extension, as selecting by

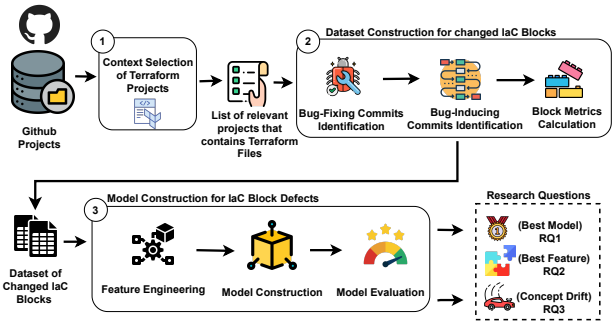

Figure 1: An overview of our study approach

Table 1: Statistics of the projects studied

Project	Creation date	LOC	# of Commits	Changed Blocks	Defective Block ratio
kubernetes-sigs/kubespray	Oct 03, 2015	30,186	7,314	1,035	16%
oracle-terraform-modules/terraform-oci-oke	Mar 22, 2019	8,866	445	1,933	16%
chanzuckerberg/cztack	Jun 27, 2018	7,874	504	1,357	20%
zenml-io/mlstacks	Jun 29, 2022	11,368	412	1,839	26%
cloudfoundry/bosh-bootloader	Jan 25, 2016	271,919	3,173	1,823	8%
cattle-ops/terraform-aws-gitlab-runner	Nov 20, 2017	2,859	977	997	25%
ministryofjustice/cloud-platform-infrastructure	Feb 23, 2018	9,406	4,679	2,409	12%
Azure/az-hop	Nov 19, 2020	11,775	3,923	1,265	5%
magma/magma	Feb 15, 2019	566,380	12,116	542	8%
GoogleCloudPlatform/cloud-foundation-fabric	May 03, 2019	66,325	4,691	5,812	23%
SUSE/ha-sap-terraform-deployments	Oct 19, 2018	16,720	2,264	6,081	10%
Azure/Avere	Aug 03, 2018	127,987	3,301	930	36%
CDCgov/prime-simplereport	Oct 08, 2020	77,432	4,118	1,545	8%
Worklytics/psoxy	Sep 21, 2021	35,207	1,922	1,536	20%
Azure/sap-automation	Nov 16, 2021	89,764	2,666	2,187	10%
aws-observability/terraform-aws-observability-accelerator	Aug 16, 2022	5,006	226	487	26%
kube-hetzner/terraform-hcloud-kube-hetzner	Jul 30, 2021	2,755	1,452	1,328	15%
PaloAltoNetworks/terraform-azurerm-vmseries-modules	Oct 1, 2020	4,858	447	962	11%
cookpad/terraform-aws-eks	Nov 15, 2019	1,161	536	535	17%

the HCL language could result in unrelated projects that use HCL for purposes other than Terraform, such as Packer [17] or Boundary [13], which are other HashiCorp Technologies.

Initially, we used the search query *"extension:tf"* to identify repositories indexed and sorted by the GitHub Code Search API. This approach produces a maximum of 1,000 indexed files during a search period [19, 101]. We used a sorting method relying on *Last Indexed Files* filter that tracks the updated files in GitHub repositories, following the study of Verdet et al. [101]. During 7 days, we executed our extension query hourly, totaling 12 daily searches. We collected 1,093 distinct projects. We filtered out 3 archived, 406 non-starred, and 93 non-licensed projects, resulting in a selection of 591 projects. To ensure the inclusion of only relevant repositories in our study, we examined the list of projects and excluded those designed for educational courses, labs, or workshops. We excluded provider repositories developed by HashiCorp contributors in GoLang, such as `hashicorp/terraform-provider-aws` [48], where Terraform files are primarily used to test provider functionalities. Through this dataset curation, we ended up with a subset of 523 projects (68 excluded). In this step, we aim to reduce the likelihood of including prototype or test projects where Terraform files serve as mere demonstrations.

Researchers caution against relying on the GitHub collected data without proper preprocessing [59, 70, 100], as many of these projects are non-relevant and may lead to conclusions instability. Based on their recommendations, we applied three criteria (*i.e.,* C1, C2, C3) to filter out projects with insufficient information on software development. Dalla et al. [38] used the previous criteria to select Ansible-based projects. Moreover, we added another criterion (*i.e.,* C4) in which to get a sufficient number of IaC files as used by Rahman et al. [81]. On the other hand, a recent report from GitHub [47] attests to a considerable increase in the use of Terraform in recent years (2022-2023). We believe that this increase may induce projects that do not promote a dynamic IaC practice (*e.g.,* rarely commits that modify IaC files). To mitigate that, we added three block-related criteria (*i.e.,* C5, C6, C7). As follows, we define the criteria used, along with their underlying rationale, and illustrate the count of projects that meet each criterion:

- **C1: Commit frequency.** The average monthly commits must be at least 7.0. C1 aims to illustrate the presence of continuous evolution. With C1, we obtained 440 projects.
- **C2: Core Contributors.** The project requires a minimum of two contributors, and their combined number of commits must constitute 80% or more of the overall contributions. C2 seeks to regulate the interaction among contributors. This criterion yielded 371 projects.
- **C3: Push event.** The project should have seen at least one push event to its main branch within the past six months. C3 seeks indications of recent activity in development. With C3, we achieved 278 projects.
- **C4: Ratio of IaC scripts.** A minimum of 11% of the project files must consist of IaC scripts. C4 seeks to demonstrate the presence of an ample number of IaC scripts. With this criterion, we attained 76 projects.
- **C5: Sufficient number of changed blocks** The project should have a minimum of 300 modified blocks distributed among various commits. C5 provides evidence of a satisfactory number of evolving Terraform blocks, which indicate frequent block modifications. This criterion selected 63 projects.
- **C6: Ratio of defects in changed blocks.** The project must have at least 5% of defects in its changed blocks. C6 ensures projects that address and resolve defective blocks. This criterion yielded 46 projects.
- **C7: Number of defective blocks in the last six months.** The project must have at least 3 defective blocks in the last six months. C7 aims to identify projects with recent and tangible defective blocks meaning that developers still face issues with Terraform blocks. With this criterion, we ended up with 19 projects.

With the aforementioned criteria, we aimed to ensure the quality and relevance of the selected projects for our study that demonstrate sufficient IaC practice in their software development. Table 1 summarizes a statistical illustration of the datasets studied.

3.2 Dataset Creation

After selecting projects containing Terraform files, our subsequent steps involve identifying and fixing commits to resolve issues within the Terraform blocks. These steps aim to label the changed blocks as defective or neutral. Furthermore, we explain how we extract the changed blocks and measure code, process, and delta metrics.

3.2.1 Fixing-Commits Identification. To do that, we systematically analyze each project's commit history, focusing on commits whose associated messages indicate bug and fault corrections. Like previous studies in defect prediction [38, 73, 75], we use a set of keywords to mark a commit as fixing, such as: *"fix"*, *"error"*, *"patch"*, and *"flaw"*. Moreover, an identified commit containing these keywords must include changes to Terraform files. Importantly, we exclude changes related to comments, white spaces, and alterations to the documentation. However, relying solely on keyword-based techniques may result in false positive commits. Meanwhile, manually validating all commits can be a time-consuming endeavor [66, 100]. To mitigate that, we incorporate a method similar to that used by Rahman et al. [78] to categorize defective files in Puppet. This technique uses the root keywords of commits that can help categorize defective files in Puppet. We analyze the root keyword of the commit message to determine if it aligns with established patterns indicative of bug-fixing actions. For example, we observed this practice in `terraform-aws-modules/terraform-aws-eks` [11], where contributors use the *"fix"* keyword as the root in their commit messages (*e.g.,* commit d2f162b [3]) when addressing corrective actions and document the bug fixes in the CHANGELOG.md file [4].

To assess the accuracy of the identification of the fixed blocks, two authors conducted a manual validation on a representative sample of 379 fixed blocks out of 24,374 changed blocks across all the studied projects, with a confidence level of 95% and a confidence interval of 5%. Similar to the study of Openja et al. [75], we carried out an independent labeling experiment by casting a vote of either "True", "False", or "Unclear" labels on the fixed blocks. While the "True" labels represent the correct fixed blocks, the "False" labels represent an incorrect fixed block identification. The "Unclear" is used when the annotator is not able to decide whether the subject block is fixed or not before the discussion between the authors. A conflict is considered when the authors disagree on the inspected instance that has different labels "True", "False", or "Unclear". Later, the authors met to resolve all the unclear (*i.e.,* being labeled as "Unclear" by at least one annotator) and disagreement cases. A key observation was that some fixes affected the `description` argument of the blocks, which was considered as documentation changes rather than functional code modifications [5], thus classifying them as false positives. The discussion resulted in the resolution of 10 unclear instances. Overall, the authors identified 37 false positive instances, while 342 were true positives. The precision of the correctly identified fixed blocks was calculated, resulting in a precision rate of 90.24%. We reduce the impact of mislabeled fixed blocks by implementing a filter that ignores the change that affects the description argument of any block.

3.2.2 Defect-Commits Identification. After identifying the set of fixing commits, we proceeded to apply the Śliwerski, Zimmermann, and Zeller (SZZ) [92] algorithm to each fixing commit within the project to identify the commit that introduced the bug fixed in the fixing commit. The algorithm analyzes the commits in reverse order, starting from the most recent and moving backward. SZZ first uses the *"git diff"* command to extract all previously modified lines from the code. Afterward, the *"git blame"* command reveals the revision and the author who made the last modification to each line in a file. This command helps determine the changes that introduced the defective lines, resulting in the bug-inducing commit. There are several variants of the SZZ algorithm in research, including AG-SZZ [63], MA-SZZ [37], and RA-SZZ [72], designed to enhance accuracy and reduce mislabeled changes. Since the original SZZ (B-SZZ [92]) can be sensitive to noise, Fan et al. [41] conducted a study to investigate the results generated by the four variants of SZZ. They showed that RA-SZZ is a robust version of SZZ. However, applying RA-SZZ relies on RefactoringMiner [99] which makes it specific for Java programs. Moreover, They found that *Meta-Change Aware* SZZ (MA-SZZ) minimizes unnecessary inspection efforts by developers regarding lines of code. MA-SZZ is an improvement on *Annotation Graph SZZ* (AG-SZZ) as it ignores non-semantic lines (such as blank or comment lines) and those concerning only format/indentation changes. MA-SZZ further refines the handling of merge and revert operations, which can lead to mislabeled changes. Consequently, we used MA-SZZ in our study to identify bug-inducing commits. Our study uses the recent pyszz package [85], which offers various SZZ implementations. By using MA-SZZ, we label a changed block in a commit as defective if it contains at least one defective line identified; otherwise, they are considered clean.

To assess the accuracy of MA-SZZ, The same two authors inspect 379 defective blocks identified by MA-SZZ. These blocks are related to the 379 fixed blocks evaluated previously. We followed the same process used previously to evaluate the fixed blocks. First, the two authors voted "True", "False", and "Unclear" for MA-SZZ results by checking whether the defective block identified by MA-SZZ is truly defective (*i.e.,* induce a bug) and is related to the fixed block. Then, they discussed any conflicts regarding the faulty blocks and the instances voted "Unclear". During the discussion, the authors resolved 7 unclear instances and all the conflicts. The evaluation process for identifying defective blocks resulted in 16 instances classified as false positives and 363 as true positives. One reason for such a false positive is that MA-SZZ could label two successive changes on the same block argument as two buggy changes. In some cases, the first change could be a true positive. We did not address this specific type of noise because it necessitated the development of an advanced tool designed to track such Terraform code changes and refactoring operations, which fall outside the scope of our current study. The precision of correctly identified buggy blocks by MA-SZZ was 95.77%.

3.2.3 Block Metrics Extraction. In this step, we present the metrics we measure for a changed block. After applying MA-SZZ and identifying the list of defective blocks in each project, we start the analysis by iterating over all commits in a given project, considering all branches. To facilitate this, we employ Pydriller [95], which is a repository analysis tool. To focus on meaningful changes and exclude routine maintenance, we ignore merges and large commits that change more than 100 files, as recommended by the McIntosh et al. [67] study. This is because large commits can be the source of noisy data. Then, for each commit, we identify the index positions of the modified lines within each modified file. Subsequently, we utilize the SonarQube HCL parser [6] to transform the Terraform file into an Abstract Syntax Tree (AST) representation, which provides a structured view of the code syntax. Finally, we extract the start and end positions of the block identifiers. These positions serve as

boundaries for at least one index of the changed lines, allowing us to pinpoint the specific code of the changed blocks.

When identifying a block change, we measure 117 features. The complete list of these features is available in an Appendix in our replication package [18]. Such features have been collected from previous work in IaC defect studies [38, 39, 78] and could be categorized into three groups:

- *HCL Code Metrics*: These are behavioural properties that describe the structure of a Terraform block. Some of these metrics are predefined and are part of the proposed IaC catalogue [39], which provides general IaC metrics to measure the maintainability of the IaC script. For example, to count the number of nested blocks, we used a recursive method that traverses the nodes of a changed block and checks their types using AST. If a node is identified as "BlockTreeImpl", which indicates a nested block, the method increments the count.
- *Process Metrics*: These metrics focus on the development process rather than the code structure. To dive into the block level, we re-implemented 21 metrics derived from previous research [38] that used process metrics for Ansible defective files. To enhance our analysis and gain deeper insight into the developer experience and historical change of the blocks, we expanded the process metrics with the types of developer experience blocks as well as the history of change in different block types (*e.g.,* resource, variable, module).
- *Delta Metrics*: These metrics capture the amount of change in a block between two successive changes. We measure the deltas for each HCL code metric. These metrics have been used in the context of traditional source code artifacts [27] and were adopted in IaC by Dalla et al. [38] to enhance the defect prediction performance.

These metrics are collected for each modified Terraform block during every project commit and stored as tabular data in a file.

3.3 Model Construction

In this section, we explain the steps to construct a defect model that can predict from a given set of metrics whether a block is defective or clean.

3.3.1 Feature Engineering. As stated in previous studies [55, 96], highly correlated metrics can negatively impact the interpretation of the DP model. To mitigate the correlation between two metrics (*i.e.,* collinearity) and the correlation across more than two metrics (*i.e.,* multicollinearity), we use Spearman's rank correlation, since it does not assume any hypothesis on the normality of the metrics [103]. To avoid manual selection after applying Spearman's rank correlation and avoid the biased decision, we use the AutoSpearman package [57] that relies on Spearman's rank correlation and Variance Inflation Factor (*i.e.,* VIF analysis) to automatically select one metric of a group of the highest correlated metrics that share the minor correlation with other metrics that are not in the group. We chose 0.7 as the Spearman correlation coefficient threshold, which is used in defects prediction studies [55, 58]. Furthermore, we set the VIF threshold at five, as used by Jiarpakdee et al. [56].

3.3.2 Model Construction. After selecting relevant features, for each project, we constructed six machine learning models, which are: Naive Bayes (NB) [106], Logistic Regression (LR) [36], Decision Tree (DT) [32], Random Forest (RF) [51], ExtraTrees (ET) [45], and Light Gradient Boosting Machine (LightGBM) [61]. These models have been widely used in software defect prediction studies [58, 64, 100]. NB and LR are baseline learners for defect prediction studies due to their speed, simplicity, and minimal data requirements. Additionally, we incorporate tree-based algorithms, known for their performance. RF comprises several decision trees, each constructed using random subsets of the dataset. On the other hand, ET builds highly randomized decision trees using the entire dataset. LightGBM, an optimized boosting algorithm, adopts a leaf-wise algorithm with depth limitation to control the growth of its internal decision trees. We utilized the implementation of these algorithms provided by `scikit-learn` [7].

Our training data are intentionally left unbalanced, as balancing techniques can bias the interpretation of DP models [46, 97]. To account for this imbalance, we configured all algorithms (except NB) to assign equal weight to positive and negative instances during training by setting the option `class_weight = "balanced"` [53]. Since DP is a binary classification task, we used a threshold to determine whether a changed block is defective. We set the threshold value to 0.5 for all our experiments as used by Bludau et al. [31]. If the measured probability of the model for a block surpasses the threshold, the changed block is buggy.

3.3.3 Model Validation. We split the datasets into training and testing subsets to validate the model's performance. We scale the training feature values within $[0, 1]$. Rahman et al. [83] have shown that increasing training data in defect prediction improves model performance, allowing models to capture historical patterns. Furthermore, McIntosh et al. [67] underscored the importance of evaluating defect prediction models using a time-based methodology, rather than relying on random-based validation, such as K-fold cross-validation. This is because the classification of changes exhibits a time-ordered pattern. McIntosh et al. [67] also suggested a 6-month validation (*i.e.,* long-term validation) to improve the performance of the models by using a cache of changes. We adopt long-term validation, using the recent six months of changed blocks as the testing set, while the remaining data serves as the training set. During the training phase, we employ TimeSeriesSplit [8]. This technique partitions the training data into equal segments, ensuring that in each iteration, all segments occurring before the one being predicted are used for training, while the segment to be predicted becomes the test set. This approach prevents any form of data leakage.

To assess the performance of the defect models, we employ a set of measures because focusing solely on a single measure can have adverse effects on others [71, 100]. These measures originate from the four possible results in binary classification, where we categorize instances into four groups: (1) True Positive (TP), which counts the defective instances correctly classified as defective; (2) True Negative (TN), which counts the non-defective instances classified as non-defective; (3) False positive (FP) counts the non-defective instances classified as defective; (4) False Negative (FN) counts the

defective instances classified as non-defective. From the literature, we select three widely used measures, which are:

- *Matthews Correlation Coefficient (MCC)*: measures the correlation between positive instances and predicted classes. It produces a coefficient with the following interpretation: +1 represents a perfect prediction, 0 indicates that there is no improvement over random guessing, and -1 signifies complete discord between the prediction and the actual instances [69, 94, 102]. MCC is defined as follows:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

- *G-mean*: calculates the geometric mean between the recall (the proportion of true positives to all actual positives) and the complement of the False Positive Rate (FPR) [71, 89, 100]. The higher is G-mean, the better is the performance. G-mean is defined as:

$$G\text{-}mean = \sqrt{\text{Recall} \times (1 - \text{FPR})}$$

$$\text{where Recall} = \frac{TP}{TP + FN}, \quad \text{FPR} = \frac{FP}{FP + TN},$$

- *Area Under Curve (AUC)*: quantifies the area under the Receiver Operator Characteristic (ROC) curve. It provides a value between 0 and 1, where 1 indicates perfect performance; the higher the AUC, the better the performance [69].

3.4 The used statistical test

In our study, performing multiple statistical comparisons is needed to (*i*) rank the different ML algorithms and (*ii*) rank the features according to their importance. To perform multiple statistical comparisons, we use the Scott-Knott-ESD (SK-ESD) test elaborated by Tantithamthavorn et al. [98]. SK-ESD extends the Scott-Knott test with normality and effect size corrections as follows:

- Normality correction: SK-ESD applies logarithmic transformation $y = log(x + 1)$ to alleviate the skewness of the data.
- Effect size correction: SK-ESD clusters the models into statistically significant groups. SK-ESD uses Cohen's d [34] to measure the effect size between different groups and merge groups having negligible effect size, *i.e.,* having $d < 0.2$. The SK-ESD allows the optimization of the likelihood of getting falsely distinct models generated from random variation.

We use the implementation of Tantithamthavorn et al. [98] since it was updated according to the recommendation of Herbold [50]

4 RESULTS

This section reports the results of our three research questions.

4.1 RQ1: Models Performance

Approach. The goal of RQ1 is to rank the different models' performance and identify the best ML model. Hence, we evaluate 6 widely-used ML classifiers namely, LightGBM, RF, ET, LR, DT, and NB (cf. Section 3.3.2) in terms of MCC, G-mean, and AUC. In addition, we added two dummy approaches as baselines to the ML algorithm to motivate the need for ML models to efficiently detect the defective blocks. Firstly, we employ a straightforward method that relies on the median churn size value (*i.e.,* Dump) for each project within the training dataset. During the testing, a block instance is flagged as defective if its churn size value exceeds the median value from the training data. Second, we incorporate another model known as `DummyClassifier`, a Random Guess (RG) model available in the `scikit-learn` library [9]. Finally, the different scores of the MLs baselines are ranked using the SK-ESD test (cf. Section 3.4).

Results. Table 2 shows the MCC, G-mean, and AUC scores for the studied ML baselines with their SK-ESD ranks for each studied project. In each project, we observe that most of the ML models perform better than the Random Guess (RG) model since we reach at least an MCC greater than 0 and an AUC higher than 0.5. This same trend is observed for the Dump technique (*i.e.,* churn size), except for one project of the 19 studied. This finding motivates the need for ML models that learn patterns from the data since dummy baselines failed to accurately identify terraform defective blocks. Furthermore, the results indicate that no ML algorithm dominates consistently the other ML algorithms in all projects. Overall, we observe that LightGBM and LR have a higher number of wins in terms of MCC, G-mean, and ROC-AUC.

It is worth mentioning that the results show that the performance of LightGBM might vary from project to project. We observe that the MCC scores of LightGBM range from 0.036 to 0.697, and the AUC scores also range from 0.52 to 0.887. In particular, we observe that for some specific projects, while LightGBM achieves a considerably low AUC of 0.5, other simpler models, such as LR, achieve an acceptable AUC score higher than 0.7. An example of such projects is the project *zenml-io/mlstacks* in which we found that LR exceeded LightGBM, achieving an AUC score of 0.8 compared to 0.54 for LightGBM. This could be related to some characteristics of these projects, which are relatively short-lived and have an age of approximately 1.5 years (*i.e.,* since 2022). Short-lived projects could have instability in their development process, and such changed blocks could be seen as noisy instances by the model, since they do not meet the learned patterns. Therefore, LightGBM can overfit noise, making it hard to generalize. A similar observation has been made by Zeng et al. [104] in which they found that simple models such as LR are beneficial and can outperform advanced techniques such as deep learning. Our results suggest the same observation for some projects previously mentioned in which LR can achieve acceptable results with improved interpretability [104]. However, in our experiments, we focus more on performance and motivate the choice of LightGBM by the fact that it can have acceptable scores on a wider range of projects (*i.e.,* achieving the highest number of wins in *MCC* and *ROC − AUC*). LightGBM is considered also to be the best-performing model compared to other baselines in terms of AUC since it achieves an AUC score greater than 0.7 in 11 out of 19 studied projects compared to 9 out of 19 for RF, 7 out of 19 for ET, 7 out of 19 for DT, 6 out of 19 for LR and 3 out of 19 for NB. Since Terraform scripts contain different block types, each representing different infrastructure elements and behaviors, it could yield heterogeneous data. Thus, we speculate that LightGBM outperforms other models due to (i) its ability to handle complex and nonlinear relationships between the dependent and independent variables and (ii) its robustness towards class imbalance due to the boosting strategy [61].

Finally, it is worth mentioning that we observe that Random Forest (RF) and Extra Trees (ET) have fewer wins (*i.e.,* 2 and 3, respectively in terms of MCC). Despite the known effectiveness of RF models in traditional defect prediction, our study investigates, to some extent, different domains since IaC syntax languages differ from traditional language programming. In addition, we focus on the changed blocks rather than the whole change (*i.e.,* commit). This aligns with Shepperd et al.'s findings [90], which emphasized that no single defect prediction technique dominates universally.

The remaining experiments (RQ2 and RQ3) are performed by using LightGBM as the default algorithm since LightGBM achieves the best performance (*i.e.,* rank 1) in terms of MCC in 6 out of the 19 projects, the best AUC in 6 projects, and the second best scores in terms of G-mean that has the highest performance in 4 projects as mentioned earlier.

RQ1 summary: *The results of RQ1 show the need for ML algorithms to predict the defects at the block level since they outperform the dummy approaches. Furthermore, The results reveal that no single ML model consistently outperforms all others. Meanwhile, we observed that LightGBM tends to be the best-performing by achieving the best MCC score on 7 occasions and having an AUC higher than 0.7 in 11 out of 19 projects.*

4.2 RQ2: Feature Importance

Approach. In this research question, we analyze the feature's importance with LightGBM to gain insights regarding the most crucial features for IaC defective block prediction. As said in the RQ1 results, we use LightGBM as the default classifier since it provides the best performance. Specifically, similar to prior works [84, 97], for each project, we calculate the permutation importance of each feature [26] using the `PermutationImportance` [10] from scikit-learn. Finally, we rank the features according to their obtained importance scores using the SK-ESD test (cf. Section 3.4).

Results. Table 3 presents the top 3 features of each dataset with their SK-ESD ranks. Figure 2 shows the ranking box plots for the top 10 most important features. The results indicate that crucial features may vary from project to project. This can be explained by the fact that every project has its own practices and community. To this end, we have the following observations:

Process metrics tend to be the most important features. Figure 2 shows the median rank for each metric across 19 projects. We observe the recent experience (`Rexp`) and the diffusion of the lines added across the changed blocks (`Dla`) emerge as the 3-Top features in all projects. Additionally, the count of block types that have changed before, categorized by type (`SimilarChange`) tends to be among the 3-Top features with a median rank lower than 7. In particular, these two features fall into the process metric category. This is consistent with the Majumder et al. [65] study, which stressed the effectiveness of process metrics in defect prediction studies. Furthermore, metrics associated with code additions contribute to model performance, as confirmed by Zeng et al. [104], who identified that logistic regression with the number of lines added as features could outperform deep learning.

Link to code metrics projects. We also noticed that HCL code metrics, such as `TmpExpr`, `DebugFunc`, and `ImplDepData`, achieve

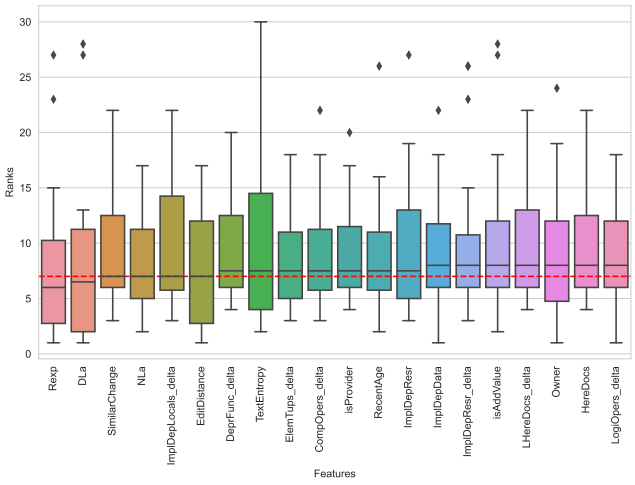

Figure 2: Box Plot of Top-20 Features by Median Rank

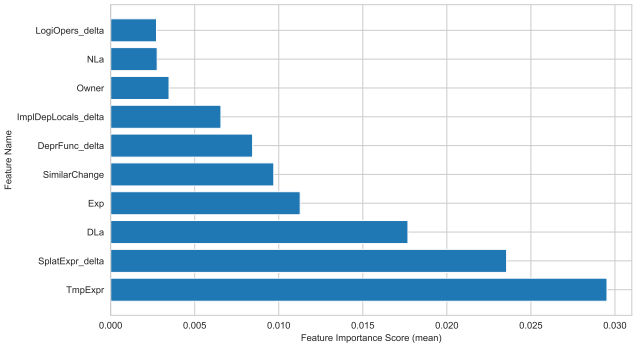

Figure 3: Top-10 Feature Importance Score of *cattle-ops/terraform-aws-gitlab-runner project*

the best ranks (*i.e.,* rank equals to 1) in 3 projects. `TmpExpr` represents the number of template expressions within the block that developers use to perform a dynamic load of values. For example, in the *cattle-ops/terraform-aws-gitlab-runner* project, the number of template expressions (referred to as `TmpExpr`) is the most significant feature, with a rank of 1. Figure 3 presents the top 10 features specific to that project based on the feature permutation technique. The `TmpExpr` code metric showed a notably high importance score. This observation aligns with the results of Opdebeeck et al. [74] in which they demonstrated that these template expressions in the IaC realm correlate with infrastructure outages when expressions are close to nullable values.

RQ2 summary: *The results of RQ2 reveal that recent experience, diffusion of added lines across changed blocks, and frequently changed block type are the Top-3 most important features. The RQ2 results suggest that process metrics tend to improve the model's performance in our study.*

Table 2: The achieved MCC, G-mean, and ROC-AUC scores with the rank (between parenthesis) using the SK-ESD test across 19 projects.

Project	MCC								G-mean								ROC-AUC							
	LightGBM	RF	ET	DT	LR	NB	Dumb	RG	LightGBM	RF	ET	DT	LR	NB	Dumb	RG	LightGBM	RF	ET	DT	LR	NB	Dumb	RG
kubernetes-sigs/kubespray	0.17 (6)	0.217 (4)	0.186 (5)	0.268 (3)	**0.351 (1)**	0.154 (7)	0.269 (2)	0.039 (8)	0.632 (4)	0.582 (5)	0.533 (6)	0.701 (2)	**0.735 (1)**	0.423 (7)	0.641 (3)	0.528 (6)	0.546 (7)	0.686 (3)	0.632 (5)	0.711 (2)	**0.759 (1)**	0.589 (6)	0.663 (4)	0.5 (8)
oracle-terraform-modules/terraform-oci-oke	0.18 (5)	0.251 (2)	0.243 (3)	0.25 (2)	0.229 (4)	**0.315 (1)**	0.164 (6)	-0.005 (7)	0.663 (5)	0.687 (3)	0.674 (4)	0.71 (2)	0.652 (6)	**0.771 (1)**	0.52 (7)	0.469 (8)	0.739 (5)	0.767 (4)	0.805 (3)	0.712 (6)	**0.847 (1)**	0.82 (2)	0.419 (8)	0.5 (7)
chanzuckerberg/cztack	0.172 (2)	0.117 (4)	0.104 (4)	0.172 (2)	0.153 (3)	-0.147 (6)	**0.243 (1)**	-0.036 (5)	0.501 (3)	0.459 (5)	0.389 (6)	0.491 (4)	**0.535 (1)**	0.315 (7)	0.527 (2)	0.469 (5)	0.653 (2)	**0.697 (1)**	0.64 (3)	0.536 (5)	0.599 (4)	0.456 (8)	0.526 (6)	0.5 (7)
zenml-io/mlstacks	0.036 (5)	0.017 (6)	0.236 (2)	-0.051 (8)	**0.321 (1)**	0.098 (4)	0.117 (3)	-0.016 (7)	0.447 (4)	0.443 (4)	0.673 (2)	0.459 (4)	**0.72 (1)**	0.279 (6)	0.327 (5)	0.484 (3)	0.544 (6)	0.568 (5)	0.731 (3)	0.495 (8)	**0.798 (1)**	0.746 (2)	0.581 (4)	0.5 (7)
cloudfoundry/bosh-bootloader	0.186 (2)	0.18 (3)	0.009 (6)	**0.299 (1)**	0.006 (6)	0.02 (5)	0.088 (4)	-0.008 (7)	0.514 (2)	0.512 (3)	0.434 (7)	**0.629 (1)**	0.5 (5)	0.462 (6)	0.504 (4)	0.476 (6)	**0.704 (1)**	0.543 (3)	0.372 (7)	0.676 (2)	0.349 (8)	0.496 (5)	0.495 (6)	0.5 (4)
cattle-ops/terraform-aws-gitlab-runner	0.226 (2)	**0.341 (1)**	0.185 (4)	-0.02 (6)	0.195 (3)	-0.038 (7)	0.061 (5)	-0.017 (6)	0.676 (4)	**0.866 (1)**	0.753 (2)	0.453 (5)	0.736 (3)	0.395 (6)	0.349 (7)	0.421 (5)	0.887 (2)	**0.923 (1)**	0.79 (3)	0.582 (6)	0.639 (4)	0.354 (8)	0.617 (5)	0.5 (7)
ministryofjustice/cloud-platform-infrastructure	**0.228 (1)**	0.112 (3)	0.129 (2)	0.051 (5)	0.092 (4)	0.036 (6)	0.032 (7)	-0.021 (8)	0.473 (4)	0.496 (3)	**0.616 (1)**	0.299 (7)	0.594 (2)	0.387 (5)	0.38 (6)	0.469 (4)	**0.77 (1)**	0.738 (2)	0.585 (5)	0.566 (7)	0.66 (3)	0.58 (6)	0.612 (4)	0.5 (8)
Azure/az-hop	0.207 (2)	**0.257 (1)**	**0.253 (1)**	0.133 (4)	0.174 (3)	-0.039 (7)	0.113 (5)	-0.012 (6)	0.73 (3)	**0.809 (1)**	0.709 (4)	0.693 (4)	0.776 (2)	0.298 (6)	0.695 (4)	0.47 (5)	0.803 (2)	**0.89 (1)**	0.799 (2)	0.786 (3)	0.771 (4)	0.466 (7)	0.575 (5)	0.5 (6)
magma/magma	**0.237 (1)**	0.206 (2)	0.19 (3)	0.012 (6)	0.106 (4)	-0.013 (7)	0.056 (5)	-0.049 (8)	**0.657 (1)**	0.62 (2)	0.633 (2)	0.465 (4)	0.583 (3)	0.355 (5)	0.452 (4)	0.445 (4)	0.665 (2)	**0.758 (1)**	0.64 (3)	0.513 (4)	0.667 (2)	0.492 (6)	0.519 (4)	0.5 (5)
GoogleCloudPlatform/cloud-foundation-fabric	0.153 (3)	0.178 (2)	**0.193 (1)**	0.115 (4)	0.101 (7)	0.107 (6)	0.113 (5)	0.002 (8)	0.608 (3)	0.632 (2)	**0.655 (1)**	0.592 (4)	0.578 (5)	0.49 (8)	0.561 (6)	0.501 (7)	0.684 (3)	0.692 (2)	**0.7 (1)**	0.63 (5)	0.627 (6)	0.637 (4)	0.593 (7)	0.5 (8)
SUSE/ha-sap-terraform-deployments	0.089 (2)	0.061 (4)	0.062 (4)	0.048 (5)	**0.111 (1)**	-0.003 (6)	0.067 (3)	-0.006 (6)	0.743 (2)	0.661 (3)	0.644 (5)	0.645 (5)	**0.78 (1)**	0.429 (6)	0.65 (4)	0.427 (6)	**0.801 (1)**	0.717 (2)	0.719 (2)	0.562 (6)	0.695 (3)	0.602 (4)	0.588 (5)	0.5 (7)
Azure/Avere	**0.299 (1)**	0.243 (2)	0.151 (3)	0.246 (2)	0.093 (4)	-0.072 (6)	-0.0 (5)	0.01 (5)	**0.645 (1)**	0.545 (4)	0.573 (3)	0.639 (2)	0.473 (6)	0.25 (7)	0.0 (8)	0.505 (5)	**0.716 (1)**	0.637 (4)	0.658 (3)	0.712 (2)	0.55 (5)	0.543 (6)	0.466 (8)	0.5 (7)
CDCgov/prime-simplereport	**0.697 (1)**	0.27 (2)	0.077 (3)	0.063 (4)	0.008 (7)	0.04 (6)	0.058 (5)	-0.037 (8)	**0.813 (1)**	0.64 (2)	0.615 (2)	0.522 (4)	0.488 (5)	0.235 (7)	0.585 (3)	0.39 (6)	**0.792 (1)**	0.683 (3)	0.66 (4)	0.58 (5)	0.442 (8)	0.56 (6)	0.789 (2)	0.5 (7)
Worklytics/psoxy	**0.183 (1)**	**0.181 (1)**	0.092 (5)	0.103 (3)	0.109 (2)	-0.015 (7)	0.095 (4)	0.01 (6)	**0.589 (1)**	0.573 (2)	0.559 (3)	0.465 (6)	0.572 (2)	0.222 (7)	0.479 (5)	0.506 (4)	**0.678 (1)**	0.659 (2)	0.624 (5)	0.592 (6)	0.63 (3)	0.626 (4)	0.56 (7)	0.5 (8)
Azure/sap-automation	0.192 (2)	0.155 (3)	0.134 (4)	**0.201 (1)**	0.126 (5)	-0.001 (7)	0.08 (6)	-0.007 (7)	0.596 (3)	0.575 (4)	0.571 (5)	**0.65 (1)**	0.612 (2)	0.408 (7)	0.489 (6)	0.483 (6)	0.653 (6)	0.681 (4)	0.661 (5)	**0.718 (1)**	0.71 (2)	0.548 (7)	0.683 (3)	0.5 (8)
aws-observability/terraform-aws-observability-accelerator	0.046 (5)	0.162 (2)	0.088 (4)	0.174 (2)	**0.32 (1)**	0.146 (3)	0.04 (5)	-0.019 (6)	0.507 (5)	0.541 (3)	0.524 (4)	0.582 (2)	**0.658 (1)**	0.525 (4)	0.376 (7)	0.488 (6)	0.52 (7)	0.597 (3)	0.541 (5)	0.621 (2)	**0.632 (1)**	0.577 (4)	0.524 (6)	0.5 (8)
kube-hetzner/terraform-hcloud-kube-hetzner	0.168 (4)	0.188 (3)	0.209 (2)	**0.214 (1)**	0.191 (3)	0.082 (5)	0.048 (6)	0.002 (7)	0.598 (5)	0.628 (4)	0.674 (2)	**0.682 (1)**	0.663 (3)	0.264 (8)	0.342 (7)	0.501 (6)	0.708 (2)	0.707 (2)	**0.729 (1)**	0.656 (4)	0.684 (3)	0.705 (2)	0.522 (5)	0.5 (6)
PaloAltoNetworks/terraform-azurerm-vmseries-modules	0.182 (2)	0.188 (2)	0.007 (6)	**0.221 (1)**	0.165 (3)	0.03 (5)	0.148 (4)	0.026 (5)	0.695 (2)	0.709 (2)	0.419 (6)	**0.784 (1)**	0.679 (3)	0.526 (5)	0.627 (4)	0.527 (5)	0.776 (2)	0.717 (3)	0.51 (7)	**0.793 (1)**	0.629 (4)	0.547 (6)	0.623 (5)	0.5 (7)
cookpad/terraform-aws-eks	**0.37 (1)**	0.33 (2)	0.22 (3)	0.317 (2)	**0.372 (1)**	0.114 (4)	0.091 (5)	0.028 (6)	0.817 (2)	0.759 (4)	0.659 (5)	0.788 (3)	**0.842 (1)**	0.408 (7)	0.338 (8)	0.524 (6)	0.819 (2)	0.821 (2)	0.819 (2)	0.804 (3)	**0.856 (1)**	0.473 (5)	0.465 (6)	0.5 (4)
Number of wins	6	3	2	4	5	1	1	0	4	2	2	4	6	1	0	0	6	3	2	2	5	0	0	0

Results in the form of $Mean(Rank)$ where $Mean$ is the mean value of performance metric i for algorithm j across all the runs and $Rank$ is the SK-ESD rank. The lower the rank the better the performance.
Algorithms with $Rank$ 1 are in bold.

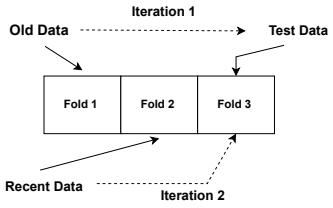

Figure 4: An overview of training models using both old and recent data

4.3 RQ3: Model Stability

Approach. To investigate potential concept drift, we evaluate the predictive performance of our approach over time using the best model from RQ1 (LightGBM). For each dataset, we divide it into three folds, arranged chronologically by the time of the commit (change). We selected three folds as the minimum number of splits to ensure that each fold contains defective blocks, as the absence of defects can impact performance measurement.

In the first iteration, we train LightGBM using the first fold (i) as the training set, representing older data, and test it on the last fold (i+2). In the second iteration, fold i+1 is used for training, representing recent data, and fold i+2 becomes the test set, as illustrated in Figure 4. We aim to identify the recurring concepts that are valid for a specific (old) fold and reappear in later folds, following prior studies [87, 93]. We repeat each iteration 11 times to account for the stochastic nature of LightGBM. Subsequently, we apply the SK-ESD Test to determine which iteration drifts. We identify a case of concept drift where the model's performance using the old data is significantly different from the model's performance when recent data are used.

Results. Table 4 reports the ranks and corresponding model performance of LightGBM models trained on two time intervals, including old and recent data, across 19 projects. In 18 out of 19 cases, we observe that the performance using old and new data differs significantly, since both scenarios have different ranks for the MCC, G-mean, and ROC-AUC scores. Our observations reveal that training on recent data outperforms training on older data, with 15 wins in MCC, 13 in G-mean, and 14 in AUC. An example of such a case is the project *aws-observability/terraform-aws-observability-accelerator*, in which we obtained a lower MCC in the old data and a slight improvement in recent training. Upon analyzing the historical change of the project, we observed that on 25 October 2022, the contributors made an update to the project, which included an update to Terraform version 1.3, involving modifications to 24 files [21]. Furthermore, the Terraform 1.3 update introduced new syntax elements to module blocks, including attributes with defaults and enhancements related to move blocks [21]. These changes aimed to enhance the extensibility and maintainability of Terraform modules by breaking the change in syntax. However, this alteration in the HCL syntax can introduce variability and instability in the change patterns associated with the blocks, making it challenging for models to identify consistent patterns.

Additionally, we have noticed that in some cases, training defect models on older or recent data can give better performance than using the entire dataset, especially when considering the results from RQ1. An example of such a case can be observed for `zenmal-io/mlstacks` and `SUSE/ha-sap-terraform-deployments` displayed improved performance with MCC values of 0.375 and 0.213, respectively, compared to training on the entire dataset, where MCC values were 0.036 and 0.089, respectively.

RQ3 summary: *Our analysis indicates that concept drift is commonly observed for Terraform defect prediction. Training on more recent data is often better than using old data.*

Table 3: The Top-3 Important features of each studied project.

Project	Feature	SK-ESD rank
kubernetes-sigs/kubespray	DLa	1
	La	2
	Exp	3
oracle-terraform-modules/terraform-oci-oke	ImplDepVars_delta	1
	Owner	2
	NetBlc	2
chanzuckerberg/cztack	La	1
	isLocals	2
	TmpExpr	3
zenml-io/mlstacks	Conds_delta	1
	NLd	2
	RecentAge	3
cloudfoundry/bosh-bootloader	DLa	1
	Kexp	2
	ImplDepResr	3
cattle-ops/terraform-aws-gitlab-runner	TmpExpr	1
	SplatExpr_delta	2
	DLa	3
ministryofjustice/cloud-platform-infrastructure	Owner	1
	ElemObjs_delta	2
	DLa	2
Azure/az-hop	Owner	1
	ImplDepData	1
	EditDistance	1
magma/magma	Bexp	1
	TextEntropy	2
	DLa	3
GoogleCloudPlatform/cloud-foundation-fabric	DebugFunc	1
	Exp	1
	ElemObjs_delta	2
SUSE/ha-sap-terraform-deployments	Bexp	1
	EditDistance	2
	IndexAccess	3
Azure/Avere	isAddDefault	1
	Rexp	2
	Kexp	3
CDCgov/prime-simplereport	Rexp	1
	NLd	2
	DLa	2
Worklytics/psoxy	EditDistance	1
	isModule	2
	SplatExpr_delta	3
Azure/sap-automation	LogiOpers_delta	1
	DLd	2
	Rexp	2
aws-observability/terraform-aws-observability-accelerator	EditDistance	1
	Rexp	2
	NLd	3
kube-hetzner/terraform-hcloud-kube-hetzner	La	1
	isLocals	1
	TextEntropy	2
PaloAltoNetworks/terraform-azurerm-vmseries-modules	DLa	1
	isAddValue	2
	CompOpers_delta	3
cookpad/terraform-aws-eks	FnCall_delta	1
	TextEntropy	2
	ElemObjs_delta	3

Table 4: The achieved MCC, G-mean, and ROC-AUC scores with the rank (between parenthesis) using SK-ESD test when varying the training time across 19 projects.

Project	MCC		G-mean		ROC-AUC	
	Old	Recent	Old	Recent	Old	Recent
kubernetes-sigs/kubespray	0.028 (2)	**0.151 (1)**	0.374 (2)	**0.549 (1)**	0.55 (2)	**0.618 (1)**
oracle-terraform-modules/terraform-oci-oke	0.111 (2)	**0.148 (1)**	0.359 (2)	**0.555 (1)**	**0.618 (1)**	0.589 (2)
chanzuckerberg/cztack	**0.145 (1)**	**0.146 (1)**	**0.662 (1)**	0.645 (2)	0.712 (2)	**0.75 (1)**
zenml-io/mlstacks	-0.022 (2)	**0.375 (1)**	0.339 (2)	**0.649 (1)**	0.46 (2)	**0.697 (1)**
cloudfoundry/bosh-bootloader	0.101 (2)	**0.154 (1)**	**0.552 (1)**	0.506 (2)	0.613 (2)	**0.653 (1)**
cattle-ops/terraform-aws-gitlab-runner	0.047 (2)	**0.134 (1)**	0.494 (2)	**0.534 (1)**	0.611 (2)	**0.651 (1)**
ministryofjustice/cloud-platform-infrastructure	**0.072 (1)**	0.047 (2)	**0.496 (1)**	0.214 (2)	**0.59 (1)**	0.504 (2)
Azure/az-hop	**0.066 (1)**	0.032 (2)	**0.305 (1)**	**0.272 (1)**	0.533 (2)	**0.582 (1)**
magma/magma	**0.041 (1)**	0.012 (2)	**0.483 (1)**	0.359 (2)	**0.686 (1)**	0.616 (2)
GoogleCloudPlatform/cloud-foundation-fabric	0.054 (2)	**0.081 (1)**	**0.513 (1)**	**0.514 (1)**	0.554 (2)	**0.577 (1)**
SUSE/ha-sap-terraform-deployments	0.043 (2)	**0.213 (1)**	0.331 (2)	**0.659 (1)**	0.665 (2)	**0.736 (1)**
Azure/Avere	**0.089 (1)**	**0.1 (1)**	**0.498 (1)**	0.43 (2)	0.555 (2)	**0.714 (1)**
CDCgov/prime-simplereport	0.093 (2)	**0.232 (1)**	0.312 (2)	**0.482 (1)**	0.678 (2)	**0.775 (1)**
Worklytics/psoxy	0.059 (2)	**0.221 (1)**	0.457 (2)	**0.642 (1)**	0.536 (2)	**0.699 (1)**
Azure/sap-automation	0.069 (2)	**0.167 (1)**	0.524 (2)	**0.584 (1)**	**0.669 (1)**	0.639 (2)
aws-observability/terraform-aws-observability-accelerator	-0.115 (2)	**0.08 (1)**	0.299 (2)	**0.539 (1)**	0.437 (2)	**0.598 (1)**
kube-hetzner/terraform-hcloud-kube-hetzner	0.136 (2)	**0.2 (1)**	0.534 (2)	**0.606 (1)**	0.603 (2)	**0.702 (1)**
PaloAltoNetworks/terraform-azurerm-vmseries-modules	**0.025 (1)**	-0.011 (2)	**0.416 (1)**	0.354 (2)	**0.683 (1)**	0.57 (2)
cookpad/terraform-aws-eks	0.065 (2)	**0.133 (1)**	0.501 (2)	**0.596 (1)**	0.695 (2)	**0.709 (1)**
Number of wins	6	15	8	13	5	14

Results in the form of $Mean(Rank)$ where $Mean$ is the mean value of performance metric i for approach j across all the runs and $Rank$ is the SK-ESD rank. The lower the rank the better the performance.
Approaches with $Rank$ 1 are in bold.

5 DISCUSSION

In this section, we discuss how the IaC community, including practitioners, tool builders, and researchers, can take advantage of our findings.

5.1 Implications for the research community

Prominent block-level Terraform defect prediction models. In RQ1, we found that we can achieve an acceptable Terraform block-level defect prediction using LightGBM. These findings suggest the need to further investigate other techniques to improve the performance of the proposed models. By doing so, developers can gain access to more accurate tools that can effectively help them determine whether a modified code block may contain defects.

Additional tool support for block-level defect prediction. In RQ3, we saw that concept drift is commonly observed in the studied projects. Therefore, it is crucial to mitigate concept drift by building tools that serve to automatically detect and mitigate concept drift. These tools can take into account major Terraform ecosystem changes and provide recommendations to developers regarding whether or not model retraining is necessary and add more developers' trust in our models.

5.2 Implications for practitioners

Maintain terraform blocks. Our RQ2 findings highlighted the importance of the added line ratio within a block, as this can directly impact the presence of defects. Developers should be aware of the number of lines added relative to the IaC file, as excessive additions can result in increased complexity and difficulties in maintenance. We recommend carefully using variables and local variables to encapsulate information without altering the behaviour of the blocks (*i.e.,* refactoring way), such as resources or modules. This approach can significantly aid in the maintenance of such sized blocks. Furthermore, our analysis in RQ2 highlights a trend in which recent experience tends to improve prediction performance. This suggests that Terraform practitioners who refrain from frequent modifications in recent days or months are more likely to introduce defects, potentially losing sight of the recent context of the block.

Mitigate the false positives in practice. From RQ1, some models can have low performance since they predict many false positive blocks. Having more false positives might lead to a higher workload for developers, since they need to investigate more blocks in practice. However, it is worth noting that the ML models are still useful, since they are discovering more defects despite the presence of false positives. Generally, The cost of not finding a defect (false negative) can be considerably higher than the developer's code inspection of a clean artefact [91]. In practice, developers can reduce the false positives by increasing the prediction probability threshold to reduce the number of blocks predicted as defective, which in some cases can lead to better performance. An example of such a case is observed for LR in the project *oracle-terraform-modules/terraform-oci-oke*. We observed that changing the threshold value of 0.5 with 0.71 can significantly improve MCC and G-mean of 0.23 and 0.65 (cf. Table 2), respectively, to 0.43, and 0.87 using 0.71 as a threshold.

Develop an online Bot. From RQ1, Our approach achieves an AUC of more than 0.7 in 11 projects. In these cases, the predictions of our approach are reliable [52]. Like the other traditional software defect prediction tools (*e.g.,* JITBot [62]), our approach can be turned into an integrated bot that can be merged as a plugin with modern software development platforms (*e.g.,* GitLab, GitHub) or as a bot in the CI/CD pipelines. It could help developers automatically identify the potentially defective blocks in the committed Terraform files.

5.3 Implications for tool builders

Stability of Terraform Environments. Our investigation, particularly in response to RQ3, shed light on the stability of the Terraform ecosystem. We have identified instances of concept drift in studied projects, indicating that patterns derived from altered code blocks may lose their validity over time. We believe this phenomenon could also be related to the frequent modifications made to the HCL syntax as well as some critical components of Terraform (*i.e.,* Providers). For example, in February 2022, the terraform-provider-aws team introduced a major release, v4.0.0, which included several breaking changes to resource and data declarations [22]. Those changes include, for example, the `aws_s3_bucket_object` resource which was deprecated, leading to infrastructure disruptions when developers attempt to store data using this resource [16]. This example aligns with the observations made in the Firefly survey [20], where they highlighted that practitioners face challenges associated with the instability of IaC tools. Hence, we recommend that tool builder be more aware of the side effects of their breaking changes on the stability of the Terraform ecosystem by following appropriate deprecation and documentation guidelines.

6 THREATS TO VALIDITY

Threats to construct validity. could be related to the threats that could create errors in our study. During the data collection, it could be that some repositories are not relevant. To mitigate that, we used seven criteria to identify repositories with sufficient information on the development process, as well as the dynamic practice of IaC. To identify the bug-fixing commits, we rely on analyzing commit messages to identify the fixing commits. This technique may lead to potential bias. To address this challenge, we employed a methodology akin to that of Rahman et al. [78], which involved analyzing commit messages and emphasizing the message headers. This method could provide information on the purpose of the commit. To discern the purpose of the commit from the message, we selected keywords from previous studies that correlate with bug-fixing commits [105]. Additionally, identifying bug-inducing commits poses another potential challenge when using the SZZ algorithm that could produce faulty labels. To mitigate this issue, we opt for the recommended variant of SZZ, namely MA-SZZ, as suggested by previous studies [37, 41], which has been shown to decrease mislabeling instances. Another potential factor could be related to *verification latency* [33], which appears when there is a delay between when a bug is induced and when it is detected by MA-SZZ. Since our primary goal is to demonstrate the ability to predict defects at the block level, we did not take verification latency into account during this study. We investigate this issue in a future study where we will be focusing more on this crucial concept before moving to the production side. Furthermore, there may be an imprecise computation of the metrics for the blocks. To address this issue in our implementation, we employ a test unit to ensure the accuracy of the metric measurements.

Threats to internal validity. could be related to the factors that were not considered and could impact the variables under investigation. In our experiments, we used a mixture of product, process, and delta metrics as suggested with a similar study for predicting faulty methods [77]. For all projects, we did not study the metrics of the subset (*e.g.,* process versus code metrics) that could affect the classification. We tried one dataset with only product metrics and observed that classifiers could dramatically lose their performance. Furthermore, such metrics in our study can be highly correlated, which may further affect the interpretability of our models. We tried AutoSpearman [57] as a feature selection technique based on correlation and redundancy analysis to mitigate that. Furthermore, our datasets are imbalanced, which could affect the performance of the models. We also tested the SMOTE as a balancing technique for two projects. However, we also observed that the SMOTE could negatively influence the model's performance. Future related studies should carefully address aspects such as feature selection and class-balancing techniques, which are outside the scope of the current study.

Threats to external validity. Our study investigated 19 open-source Terraform-based projects, which may not be generalized to all Terraform-based projects. To mitigate that, we study relevant, widely-used projects from different GitHub organizations to minimize potential bias in our findings. However, replication of our study using more projects could provide valuable insights.

7 CONCLUSION

This paper investigated defect prediction at the Just-In-Time block level in Terraform files, a widely utilized Infrastructure-as-Code (IaC) provisioning tool. Our experiments were carried out on 19 Terraform-based open-source projects gathered from GitHub, using six ML algorithms to learn and test defective blocks over the last six months. In summary, the study results demonstrate the capability of ML algorithms to predict defects at the block level, with LightGBM achieving an average MCC of 0.21 and an AUC score of 0.71. In particular, the relative added lines, recent developer experience, and frequently changed block types were identified as the top three features with the lowest median rank. Our findings also revealed that defect prediction models become less effective over time, losing their ability to predict the nature of blocks.

For future work, we aim to enhance our models' performance by considering other pertinent aspects and expanding our experiments to encompass a more extensive range of projects. This initiative seeks to provide valuable insights on defective blocks to the IaC research community and practitioners. Moreover, the results suggest that interpretable models like logistic regression can reach acceptable performance, we aim to investigate further the trade-off between the different ML algorithms' performance and interpretability. In addition, we plan to deploy our models online and collect feedback from IaC practitioners.

REFERENCES

[1] [n. d.]. https://registry.terraform.io/modules/terraform-aws-modules/eks/aws. Accessed on: Nov 1, 2023.
[2] [n. d.]. https://survey.stackoverflow.co/2023/#most-popular-technologies-tools-tech-prof. Accessed on: Nov 1, 2023.
[3] [n. d.]. https://github.com/terraform-aws-modules/terraform-aws-eks/commit/d2f162b190596756f1bc9d8f8061e68329c3e5c4. Accessed on: Nov 1, 2023.
[4] [n. d.]. https://github.com/terraform-aws-modules/terraform-aws-eks/blob/master/CHANGELOG.md. Accessed on: Nov 1, 2023.
[5] [n. d.]. https://developer.hashicorp.com/terraform/language/values/variables#input-variable-documentation. Accessed on: Nov 1, 2023.
[6] [n. d.]. https://github.com/SonarSource/sonar-iac/tree/master/iac-extensions/terraform. Accessed on: Nov 1, 2023.
[7] [n. d.]. https://scikit-learn.org/stable/. Accessed on: Nov 1, 2023.
[8] [n. d.]. https://scikit-learn.org/stable/modules/generated/sklearn.model.TimeSeriesSplit.html. Accessed on: Nov 1, 2023.
[9] [n. d.]. https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html. Accessed on: Dec 24, 2023.
[10] [n. d.]. https://scikit-learn.org/stable/modules/permutation_importance.html. Accessed on: Nov 1, 2023.
[11] [n. d.]. AWS EKS Terraform module. https://github.com/terraform-aws-modules/terraform-aws-eks. Accessed on: Nov 1, 2023.
[12] [n. d.]. AWS Lambda. https://aws.amazon.com/lambda/. Accessed on: Oct 1, 2023.
[13] [n. d.]. Boundary. https://www.boundaryproject.io/. Accessed on: Oct 2, 2023.
[14] [n. d.]. Crane: Uber's Next-Gen Infrastructure Stack. https://www.uber.com/en-IN/blog/crane-ubers-next-gen-infrastructure-stack/. Accessed on: Sept 23, 2023.
[15] [n. d.]. GitHub REST API documentation. https://docs.github.com/en/rest?apiVersion=2022-11-28. Accessed on: Oct 1, 2023.
[16] [n. d.]. No way to migrate from aws_s3_bucket_object to aws_s3_object. https://github.com/hashicorp/terraform-provider-aws/issues/25412. Accessed on: Oct 14, 2023.
[17] [n. d.]. Packer. https://www.packer.io, note = Accessed on: Oct 2, 2023.
[18] [n. d.]. Replication Package for the paper: "Fine-Grained Just-In-Time Defect Prediction at the Block Level in Infrastructure-as-Code (IaC)". https://figshare.com/s/facf0404c96b32274a35. Accessed on: Dec 22, 2023.
[19] [n. d.]. Repository Contents. https://docs.github.com/en/rest/repos/contents?apiVersion=2022-11-28. Accessed on: Oct 1, 2023.
[20] [n. d.]. State of Infrastructure-as-Code 2023. https://www.firefly.ai/state-of-iac. Accessed on: Sept 23, 2023.
[21] [n. d.]. Terraform 1.3 Improvement. https://www.hashicorp.com/blog/terraform-1-3-improves-extensibility-and-maintainability-of-terraform-modules. Accessed on: Nov 12, 2023.
[22] [n. d.]. Terraform AWS Provider Version 4 Upgrade Guide. https://registry.terraform.io/providers/hashicorp/aws/latest/docs/guides/version-4-upgrade#resource-aws_s3_bucket_object. Accessed on: Oct 1, 2023.
[23] [n. d.]. Terraform Configuration Syntax. https://developer.hashicorp.com/terraform/docs. Accessed on: Sept 26, 2023.
[24] [n. d.]. Terraform Configuration Syntax. https://developer.hashicorp.com/terraform/language/syntax/configuration. Accessed on: Sept 26, 2023.
[25] [n. d.]. Using Terraform to Manage Infrastructure. https://shopify.engineering/manage-infrastructure-with-terraform. Accessed on: Sept 23, 2023.
[26] Andre Altmann, Laura Tolosi, Oliver Sander, and Thomas Lengauer. 2010. Permutation importance: A corrected feature importance measure. *Bioinformatics (Oxford, England)* 26 (04 2010), 1340–7.
[27] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. 2010. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software* 83, 1 (2010), 2–17. SI: Top Scholars.
[28] Google Cloud Load Balancers. [n. d.]. Google Cloud Load Balancers. https://cloud.google.com/load-balancing. Accessed on: Oct 1, 2023.
[29] Akash Banginwar. [n. d.]. Same issue. https://github.com/terraform-aws-modules/terraform-aws-eks/issues/950#issuecomment-1015847431. Accessed on: Oct 1, 2023.
[30] Mahi Begoug, Narjes Bessghaier, Ali Ouni, Eman Alomar, and Mohamed Wiem Mkaouer. 2023. What Do Infrastructure-as-Code Practitioners Discuss: An Empirical Study on Stack Overflow. In *Proceedings of the 17th International Conference on Empirical Software Engineering and Measurement (ESEM '23)*. 11 pages.
[31] Peter Bludau and Alexander Pretschner. 2022. Feature sets in just-in-time defect prediction: an empirical evaluation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*. 22–31.
[32] L. Breiman, Jerome H. Friedman, Richard A. Olshen, and C. J. Stone. 1984. Classification and Regression Trees. *Biometrics* 40 (1984), 874. https://api.semanticscholar.org/CorpusID:29458883
[33] George G Cabral, Leandro L Minku, Emad Shihab, and Suhaib Mujahid. 2019. Class imbalance evolution and verification latency in just-in-time software defect prediction. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 666–676.
[34] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences*. Academic press.
[35] Hashi Corp. [n. d.]. Terraform. https://www.terraform.io. Accessed on: Oct 1, 2023.
[36] David R Cox. 1958. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)* 20, 2 (1958), 215–232.
[37] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. 2017. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering* 43, 7 (2017), 641–657.
[38] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Tamburri. 2021. Within-Project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics. *IEEE Transactions on Software Engineering* PP (01 2021), 1–1.
[39] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. 2020. Toward a catalog of software quality metrics for infrastructure code. *Journal of Systems and Software* 170 (2020), 110726.
[40] Azure Active Directory. [n. d.]. Azure Active Directory. https://azure.microsoft.com/en-us/products/active-directory-ds. Accessed on: Oct 1, 2023.
[41] Yuanrui Fan, Xin Xia, Daniel Alencar Da Costa, David Lo, Ahmed E Hassan, and Shanping Li. 2019. The impact of mislabeled changes by szz on just-in-time defect prediction. *IEEE transactions on software engineering* 47, 8 (2019), 1559–1586.
[42] AWS Cloud Formation. [n. d.]. Cloud Formation. https://aws.amazon.com/cloudformation/. Accessed on: Oct 1, 2023.
[43] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. 2014. An Empirical Study of Just-in-Time Defect Prediction Using Cross-Project Models. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 172–181.
[44] Jaime Hidalgo García. [n. d.]. Cycle error on destroy when updating from 12.0 to 12.2. terraform-aws-modules/terraform-aws-eks/issues/950. Accessed on: Oct 1, 2023.
[45] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely randomized trees. *Machine Learning* 63 (2006), 3–42. https://api.semanticscholar.org/CorpusID:15137276
[46] Görkem Giray, Kwabena Ebo Bennin, Ömer Köksal, Önder Babur, and Bedir Tekinerdogan. 2023. On the use of deep learning in software defect prediction. *Journal of Systems and Software* 195 (2023), 111537.
[47] GitHub. [n. d.]. The State of Open Source and AI. https://github.blog/2023-11-08-the-state-of-open-source-and-ai/. Accessed on: Nov 12, 2023.
[48] Hashi Corp. [n. d.]. https://github.com/hashicorp/terraform-provider-aws. Accessed on: Nov 1, 2023.
[49] Red Hat. [n. d.]. Ansible. https://www.ansible.com/. Accessed on: Oct 1, 2023.
[50] Steffen Herbold. 2017. Comments on ScottKnottESD in response to" An empirical comparison of model validation techniques for defect prediction models". *IEEE Transactions on Software Engineering* 43, 11 (2017), 1091–1094.
[51] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, Vol. 1. IEEE, 278–282.
[52] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. 2013. *Applied logistic regression*. Vol. 398. John Wiley & Sons.
[53] Khairul Islam, Toufique Ahmed, Rifat Shahriyar, Anindya Iqbal, and Gias Uddin. 2022. Early prediction for merged vs abandoned code changes in modern code reviews. *Information and Software Technology* 142 (2022), 106756.
[54] Yujuan Jiang and Bram Adams. 2015. Co-evolution of Infrastructure and Source Code - An Empirical Study. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 45–55.
[55] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Ahmed E. Hassan. 2021. The Impact of Correlated Metrics on the Interpretation of Defect Models. *IEEE Transactions on Software Engineering* 47, 2 (2021), 320–331.
[56] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Akinori Ihara, and Kenichi Matsumoto. 2016. A Study of Redundant Metrics in Defect Prediction Datasets. In *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 51–52.
[57] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Christoph Treude. 2018. AutoSpearman: Automatically Mitigating Correlated Software Metrics for Interpreting Defect Models. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 92–103.
[58] Jirayus Jiarpakdee, Chakkrit Kla Tantithamthavorn, Hoa Khanh Dam, and John Grundy. 2022. An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering* 48, 1 (2022), 166–185.
[59] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub

(MSR 2014). Association for Computing Machinery, New York, NY, USA, 92–101.

[60] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.

[61] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017), 3146–3154.

[62] Chaiyakarn Khanan, Worawit Luewichana, Krissakorn Pruktharathikoon, Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, and Thanwadee Sunetnanta. 2020. JITBot: an explainable just-in-time defect prediction bot. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 1336–1339.

[63] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. 2006. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 81–90.

[64] Masanari Kondo, Cor-Paul Bezemer, Yasutaka Kamei, Ahmed E. Hassan, and Osamu Mizuno. 2019. The Impact of Feature Reduction Techniques on Defect Prediction Models. *Empirical Softw. Engg.* 24, 4 (aug 2019), 1925–1963.

[65] Suvodeep Majumder, Pranav Mody, and Tim Menzies. 2022. Revisiting Process versus Product Metrics: A Large Scale Analysis. *Empirical Softw. Engg.* 27, 3 (may 2022), 42 pages.

[66] Everton da S. Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical Debt. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. 9–15.

[67] Shane McIntosh and Yasutaka Kamei. 2018. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. *IEEE Transactions on Software Engineering* 44, 5 (2018), 412–428.

[68] Audris Mockus and David Weiss. 2002. Predicting risk of software changes. *Bell Labs Technical Journal* 5 (06 2002), 169–180.

[69] Rebecca Moussa and Federica Sarro. 2022. On the Use of Evaluation Measures for Defect Prediction Studies. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 101–113.

[70] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating github for engineered software projects. *Empirical Software Engineering* 22 (2017), 3219–3253.

[71] Shrikanth N.C., Suvodeep Majumder, and Tim Menzies. 2021. Early Life Cycle Software Defect Prediction. Why? How?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 448–459.

[72] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2018. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 380–390.

[73] Chao Ni, Xin Xia, David Lo, Xiaohu Yang, and Ahmed E. Hassan. 2022. Just-In-Time Defect Prediction on JavaScript Projects: A Replication Study. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 76 (aug 2022), 38 pages.

[74] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2022. Smelly variables in ansible infrastructure code: detection, prevalence, and lifetime. In *19th International Conference on Mining Software Repositories*. 61–72.

[75] Moses Openja, Mohammad Mehdi Morovati, Le An, Foutse Khomh, and Mouna Abidi. 2022. Technical debts and faults in open-source quantum software systems: An empirical study. *Journal of Systems and Software* 193 (2022), 111458.

[76] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software* 150 (2019), 22–36.

[77] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. On the Performance of Method-Level Bug Prediction: A Negative Result. *Journal of Systems and Software* 161 (12 2019), 110493.

[78] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. 2020. Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 752–764.

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

[80] 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)*. 164–175.

[81] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. 2021. Security smells in ansible and chef scripts: A replication study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2021), 1–31.

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

[83] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. 2013. Sample size vs. bias in defect prediction. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. 147–157.

[84] Gopi Krishnan Rajbahadur, Shaowei Wang, Gustavo A Oliva, Yasutaka Kamei, and Ahmed E Hassan. 2021. The impact of feature importance methods on the interpretation of defect classifiers. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2245–2261.

[85] Giovanni Rosa. [n. d.]. pyszz. https://github.com/grosa1/pyszz. Accessed on: Nov 12, 2023.

[86] Nuno Saavedra and João Ferreira. 2023. GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code. 1–12.

[87] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. 2022. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering* 29, 1 (2022), 21.

[88] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *13th International Conference on Mining Software Repositories*. 189–200.

[89] Raed Shatnawi. 2010. A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems. *IEEE Transactions on Software Engineering* 36, 2 (2010), 216–225.

[90] Martin Shepperd, David Bowes, and Tracy Hall. 2014. Researcher Bias: The Use of Machine Learning in Software Defect Prediction. *IEEE Transactions on Software Engineering* 40, 6 (2014), 603–616.

[91] Swapnil Shukla, T Radhakrishnan, K Muthukumaran, and Lalita Bhanu Murthy Neti. 2018. Multi-objective cross-version defect prediction. *Soft Computing* 22 (2018), 1959–1980.

[92] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.

[93] Liyan Song, Leandro Minku, Cong Teng, and Xin Yao. 2023. A Practical Human Labeling Method for Online Just-in-Time Software Defect Prediction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 605–617.

[94] Qinbao Song, Yuchen Guo, and Martin Shepperd. 2018. A Comprehensive Investigation of the Role of Imbalanced Learning for Software Defect Prediction. *IEEE Transactions on Software Engineering* PP (05 2018), 1–1.

[95] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python Framework for Mining Software Repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 908–911.

[96] Chakkrit Tantithamthavorn and Ahmed E. Hassan. 2018. An Experience Report on Defect Modelling in Practice: Pitfalls and Challenges. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 286–295.

[97] Chakkrit Tantithamthavorn, Ahmed E. Hassan, and Kenichi Matsumoto. 2020. The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models. *IEEE Transactions on Software Engineering* 46, 11 (2020), 1200–1219.

[98] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2018. The Impact of Automated Parameter Optimization for Defect Prediction Models. (2018).

[99] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2020), 930–950.

[100] Huy Tu, Zhe Yu, and Tim Menzies. 2022. Better Data Labelling With EMBLEM (and how that Impacts Defect Prediction). *IEEE Transactions on Software Engineering* 48, 1 (2022), 278–294.

[101] Alexandre Verdet, Mohammad Hamdaqa, Leuson Da Silva, and Foutse Khomh. 2023. Exploring Security Practices in Infrastructure as Code: An Empirical Study. *arXiv preprint arXiv:2308.03952* (2023).

[102] Jingxiu Yao and Martin Shepperd. 2020. Assessing Software Defection Prediction Performance: Why Using the Matthews Correlation Coefficient Matters. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering* (Trondheim, Norway) *(EASE '20)*. Association for Computing Machinery, New York, NY, USA, 120–129.

[103] Jerrold H Zar. 2005. Spearman rank correlation. *Encyclopedia of Biostatistics* 7 (2005).

[104] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep Just-in-Time Defect Prediction: How Far Are We?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 427–438.

[105] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2014. Towards Building a Universal Defect Prediction Model. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 182–191.

[106] Harry Zhang. 2004. The Optimality of Naive Bayes. *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2004* 2.