Replication of “Source Code Properties of Defective Infrastructure as Code Scripts” (Rahman & Williams, 2019)

1st MIGUEL CARRASCO GUIRAO
Barcelona, Spain
miguel.carrasco-guirao@polymtl.ca

2nd POL MARGARIT FISAS
Barcelona, Spain
pol.margarit-i-fisas@polymtl.ca

3rd ALI ENTEZARI
Montréal, Canada
ali.entezari@polymtl.ca

4th OHTO SYSILÄ
Montréal, Canada
ohto-viljami.sysila@etud.polymtl.ca

***Abstract*—This paper reports a replication of key parts of Rahman and Williams (2019). The replication focuses on (i) reproducing subsections 3.1.1 (Repository Collection) and 3.1.2 (Commit Message Processing) of the original methodology and (ii) answering research questions RQ1 and RQ3 using the authors' released datasets. We describe our mining setup with the GitHub API and caching to respect rate limits, operationalize commit message processing to build extended commit messages, and then use the supplied datasets to analyze source code properties of Puppet scripts (RQ1) and to build defect prediction models (RQ3). We report our findings, compare them against the original study, and discuss deviations and threats to validity.**

***Index Terms*—Replication, Mining Software Repositories, Infrastructure as Code, Puppet, Defect Prediction, Empirical Software Engineering**

I. Introduction

Infrastructure as Code (IaC) enables automated, reliable deployment pipelines. Defects in IaC scripts can undermine these pipelines. Rahman and Williams (2019) investigated which source code properties of Puppet scripts correlate with defectiveness and how those properties can be used to build defect prediction models.

In this project, we replicate part of their study by reproducing Sections 3.1.1 (Repository Collection) and 3.1.2 (Commit Message Processing), and by addressing two research questions: RQ1, which investigates what source code properties characterize defective IaC scripts, and RQ3, which explores how defect prediction models can be constructed using those properties. Our replication relies on the authors' released datasets and compares our findings against the original results.

II. Background

Infrastructure as Code (IaC) is the practice of provisioning and managing computing infrastructure through machine-readable files rather than ad-hoc processes. IaC enables deployment automation, consistency across environments, and introduction of changes to infrastructure into the same application version control and review cycle as non-cloud software development. Treating infrastructure specifications as code enables teams to leverage automated testing, continuous integration, and formal reuse.

Among the tools supporting IaC, *Puppet* is a leading configuration management tool. Puppet scripts, written in the `.pp` programming language, state system resources such as files, packages, and services, and describe how such resources are to be installed or configured. Puppet supports modularization by classes and modules, and allows practitioners to encapsulate dependencies and constraints as explicit source code. Because of the popularity of Puppet and its availability as an open source, it was at the center of the first investigation.

In this context, a *defect* in an IaC script refers to a flaw that needs to be fixed or replaced. Defects may arise due to wrong configuration values, misplaced file paths, improper permissions, missing dependencies, or other implementation errors that compromise the integrity of automated deployment pipelines. Defects can have cataclysmic consequences: for example, Rahman and Williams report an incident in which a buggy Puppet script wiped out home directories of hundreds of Wikimedia users.

Rahman and Williams (2019) investigated systematically what source code features of IaC scripts are statistically correlated with defectiveness. Using evidence from four large open-source ecosystems (Mirantis, Mozilla, OpenStack, and Wikimedia Commons), they identified twelve measurable features, e.g., *lines of code*, *hard-coded strings*, and *include* statements, that are statistically correlated with defective scripts. They followed these findings up with a practitioners' survey and constructed defect prediction models using typical statistical learners. Their models achieved promising levels of precision and recall, suggesting the empirical utility of source code features for defect prediction in IaC.

For this replication, we focus on two of their research questions:

- **RQ1:** What source code properties characterize defective IaC scripts?
- **RQ3:** How can defect prediction models be constructed using those properties?

We selected these two research questions because they are directly supported by the authors' publicly released datasets and provide a clear basis for comparing our replicated results against the original study.

III. Research Questions

The original study by Rahman and Williams (2019) defined three research questions (RQ1–RQ3). In this replication, we address only RQ1 and RQ3, as they can be directly investigated using the authors' publicly released datasets. RQ2, which required surveying practitioners, was excluded because it depended on external data collection that is outside the scope of this replication exercise.

- **RQ1:** What source code properties characterize defective Infrastructure as Code (IaC) scripts?
- **RQ3:** How can defect prediction models be constructed using the identified source code properties?

By focusing on these two questions, our replication emphasizes both the identification of defect-related source code properties and the evaluation of predictive models that make use of them.

IV. Methodology

This section describes the process followed to replicate the repository mining phase and implement the filtering restrictions defined by Rahman and Williams (2019). All steps were automated in Python 3 using the GitHub REST API, ensuring reproducibility and independence from manual dataset curation.

A. Repository Mining Setup

Repository data were collected automatically from GitHub. Each repository was queried via the REST API to obtain metadata and commit history. This approach was chosen to maintain transparency and to enable other researchers to replicate the same pipeline.

Only public repositories were considered (Restriction R1), as these allow full verification of the results and comply with open science principles. Private or archived repositories were excluded to ensure accessibility and data consistency.

B. Restriction R2 — Puppet File Ratio

To ensure that repositories were primarily Puppet-based, Restriction R2 required that at least 11% of the files in the repository be Puppet scripts (`.pp` files). The computation was based on the number of files rather than file size or lines of code. This decision was made for three reasons:

1) File counts are independent of script length, avoiding bias toward larger files that may not contain meaningful IaC logic.
2) Counting by files instead of bytes prevents the inclusion of large, non-source assets (e.g., binaries, documentation, or templates) that distort the ratio.
3) This method preserves alignment with the original study, which also defined R2 in terms of file counts.

For efficiency and to avoid GitHub API rate limits, repositories were cloned locally using a shallow clone (`git clone --depth 1`). This retrieves only the latest snapshot of the repository without full history, reducing both time and bandwidth while maintaining a complete view of the file structure.

C. Restriction R3 — Repository Activity

The original paper informally mentions "two commits per month", but the wording is ambiguous as to whether this must hold for every calendar month or only on average over a period. In this replication we make the assumption explicit and operationalize R3 as an *average* of at least two commits per month over the last 24 full months (excluding the current month), rather than enforcing the threshold for each individual month. This adjustment was made to reflect realistic development patterns in open-source projects, which often experience uneven commit activity due to vacations, release cycles, or temporary inactivity.

The justification for this modification is threefold:

1) Commit activity is inherently bursty, and a strict monthly threshold may unfairly exclude active projects.
2) The average-based approach captures long-term project vitality while tolerating natural fluctuations.
3) Using a 24-month window provides a representative temporal scope of project activity without overweighting historical data.

The current month was excluded from calculations since it may contain incomplete data. Commits were collected via paginated API requests (100 commits per page) until reaching 24 months of history or the beginning of the repository's activity.

D. Commit Collection and Storage

For repositories satisfying R1–R3, all commits were extracted with their SHA, author date, and first-line message. These commits were serialized into a JSON structure and stored locally for reproducibility. This design allows later verification of data integrity and potential reuse for further analysis, such as the classification of defect-inducing commits.

E. Analysis Scope

While the mining pipeline successfully reproduces the filtering and collection logic of Rahman and Williams (2019), the datasets used to address RQ1 and RQ3 were obtained directly from the authors' released data. This ensures that the statistical and machine learning analyses are consistent with the original study while verifying the reproducibility of the mining process itself. The data collected through our own mining implementation were not used in the analyses, as this step was purely formative and optional, and would have additionally required extensive preprocessing and message filtering to match the original dataset's structure.

F. Defect Prediction Models (RQ3)

To replicate the construction of defect prediction models, we followed the same approach as Rahman and Williams (2019). Each Puppet script in the dataset is represented by the counts of the twelve source code properties identified through qualitative analysis (e.g., lines of code, hard-coded strings, include statements). Numerical features were log-transformed to reduce skewness, and principal component analysis (PCA) was used to address potential multicollinearity among properties. Following the original setup, we selected the minimum number of principal components that explained at least 95% of the total variance.

Using these transformed features, we trained five common statistical learners, all implemented with `scikit-learn`:

1) Classification and Regression Trees (CART),
2) k-Nearest Neighbors (KNN),
3) Logistic Regression (LR),
4) Naive Bayes (NB),
5) Random Forest (RF).

Models were evaluated using stratified 10×10 cross-validation to provide stable estimates of generalization performance. The following performance measures were computed: precision, recall, F1 score, and the area under the ROC curve (AUC). To assess the statistical significance of performance differences, we applied the Dunn test with Holm correction for pairwise comparisons of AUC values between learners.

This setup allows us to reproduce RQ3 from the original study: namely, whether source code properties can be used effectively to build defect prediction models for IaC scripts, and which learners perform best across datasets.

V. Results

A. Repository Mining Phase

The mining phase was executed using the Python pipeline described in the previous section. Initially, the repositories referenced in Rahman and Williams (2019) were analyzed, specifically the seven OpenStack Puppet projects originally included in their study:

- openstack/puppet-keystone
- openstack/puppet-nova
- openstack/puppet-neutron
- openstack/puppet-glance
- openstack/puppet-cinder
- openstack/puppet-horizon
- openstack/puppet-swift

The following execution summary was obtained when running the mining script:

```
Checking openstack/puppet-keystone...
openstack/puppet-keystone: <11% of .pp
Checking openstack/puppet-nova...
openstack/puppet-nova: <11% of .pp
Checking openstack/puppet-neutron...
openstack/puppet-neutron: <11% of .pp
Checking openstack/puppet-glance...
Checking openstack/puppet-cinder...
Checking openstack/puppet-horizon...
openstack/puppet-horizon: <11% of .pp
Checking openstack/puppet-swift...
Commits saved in output/mined_commits.json
```

From this execution, only `openstack/puppet-swift` met all restrictions (R1–R3) and was successfully mined. The rest were excluded because they no longer satisfy Restriction R2 (less than 11% of files are Puppet scripts) or, in some cases, exhibit very low or irregular commit activity that violates Restriction R3.

In addition, we attempted to reproduce the mining process using the alternative repositories provided in the course materials (e.g., `mozilla/gecko-dev`, `wikimedia/mediawiki`, `Mirantis/kubernetes-release`), but none of these projects satisfied the Puppet ratio or activity thresholds either. These repositories use different configuration management systems or contain mixed languages, leading to automatic exclusion by our filters.

This confirms that, while the mining pipeline itself is functional and consistent with the logic of Rahman and Williams (2019), most of the original and educational repositories are no longer valid candidates for replication due to ecosystem changes.

B. Defect Prediction Models (RQ3)

Table I summarizes the median scores obtained from our 10×10 cross-validation experiments. Across datasets, Logistic Regression (LR), Naive Bayes (NB), and Random Forest (RF) consistently outperformed CART, with AUC values typically above 0.70. The best performing models varied slightly by dataset: RF achieved the highest AUC on MOZ (0.81) and OST (0.76), NB was strongest on WIK (0.79), and LR performed best on MIR (0.75). Pairwise Dunn tests confirmed that these differences were statistically significant in several cases, particularly when comparing RF against CART or KNN.

TABLE I
Median prediction performance over 10×10 CV

Dataset	Model	AUC	Precision	Recall
MIR	LR	0.75	0.70	0.67
MIR	RF	0.74	0.69	0.70
MOZ	RF	0.81	0.72	0.73
OST	RF	0.76	0.72	0.79
WIK	NB	0.79	0.76	0.69

Compared to the original study, our replication finds in general consistent patterns: Random Forest and Logistic Regression possess the most perfect defect prediction capability, and CART and KNN are less competitive. AUC values likewise fall within the same range (0.70–0.80) that Rahman and Williams reported, demonstrating that source code attributes alone are powerful predictors of buggy IaC scripts. One of the contrasts we observed is Naive Bayes surprisingly performing good on the WIK dataset, whereas the original paper reported RF and LR as the top performers across all the datasets.

This difference may be brought about by differences in data preprocessing or changes in dataset patterns since the first release.

In summary, the results support the conclusion that defect prediction models built on simple source code features may possess acceptable accuracy, and that ensemble methods such as Random Forest provide the most robust performance across a range of datasets.

VI. Discussion

A. Repository Mining Discussion

The mining experiment reveals that most repositories originally analyzed by Rahman and Williams (2019) no longer comply with the same restrictions (R2 and R3). The main causes are: (1) project archival or migration from Puppet to newer frameworks such as Ansible or YAML-based pipelines, (2) a significantly reduced proportion of Puppet files in mixed-language repositories, and (3) decreased development activity leading to violation of the minimum commit frequency requirement.

Moreover, the repositories provided as examples for the course exercise (e.g., Mozilla, Wikimedia, and Mirantis) also failed to meet the criteria, either because they do not primarily use Puppet or because their activity levels are inconsistent with the R3 threshold. This further emphasizes the difficulty of reproducing historical studies when the technological landscape has evolved.

The decision to measure the Puppet ratio (R2) by file count rather than file size proved reliable and allowed for objective filtering, although the 11% threshold is increasingly restrictive in modern multi-language repositories. The adaptation of Restriction R3 to an average-based metric was also validated, as open-source projects typically display bursty commit behavior with periods of inactivity.

Overall, the mining phase successfully replicates the filtering logic of the original study but shows that reproducing the exact dataset is no longer feasible. The only repository satisfying all constraints, `openstack/puppet-swift`, confirms that the implemented restrictions and mining logic behave correctly, even if the underlying ecosystem has evolved.

B. Defect Prediction Models Discussion

Our results for RQ3 largely confirm the findings of Rahman and Williams (2019). Random Forest and Logistic Regression are the most stable learners to predict defective IaC scripts with AUC consistently between 0.70–0.80 in the replicated as well as the original study. This means that relatively simple statistical learners, trained on source code property counts, can achieve the same predictive performance as more complex text-based or process-based approaches. In our replication, Random Forest performed best for the MOZ and OST datasets, and Logistic Regression for MIR, while Naive Bayes surprisingly performed best of all on WIK. The second deviation stresses that performance differences are occasionally dataset-dependent and susceptible to data characteristics such as class balance or feature distribution.

Note that our mining data were not reused for model training. Our pipeline for reproducing repository collection and commit filtering was an exploratory step for checking the validity of the early limitations (R1–R3). However, replicating the whole defect labeling process would have required large-scale manual or semi-automated classification of commits, the same way it was performed for the SZZ approach in the original paper. For the sake of comparability and reproducibility, we fell back to the publicly available datasets provided by Rahman and Williams, which already constitute defect labels.

Reproducibility-wise, there were several challenges. First, the original repositories have evolved substantially: some have abandoned Puppet or have varying levels of activity in comparison to 2019, which makes re-mining difficult. Second, defect labeling depends on commit message interpretation and cross-validation with issue trackers. This leads to subjectivity and, even if automated using SZZ-style heuristics, can create variations across replications. Finally, subtle differences in preprocessing steps (e.g., feature extraction, log transformations, or PCA thresholds) can lead to shifts in which learners perform best, as observed in our results for WIK.

Despite these challenges, the general trend remains robust: models trained on simple source code properties can achieve reasonable accuracy, and ensemble methods such as Random Forest are a strong default choice for practitioners. Our reproduction affirms the validity of property-based defect prediction, while also demonstrating the vulnerability of full reproducibility as datasets and ecosystems evolve.

VII. Conclusion

This reproduction tried to reproduce core elements of Rahman and Williams (2019), covering repository mining and research questions RQ1 and RQ3. We could reproduce the repository collection and filtering logic of the original paper, and it was confirmed that the accessibility-based, Puppet file ratio-based, and commit activity-based criteria can be operationalized. Our mining reconfirmed that the empirical approach is sound, but also revealed that many original repositories no longer satisfy the given constraints, primarily due to migration away from Puppet and reduced activity. This identifies a general problem in empirical software engineering: earlier studies can become difficult to replicate as ecosystems evolve.

With the publicly available datasets, we replicated the analysis of source code features and defect correlation (RQ1), as well as constructing defect prediction models (RQ3). Our statistical tests confirmed that features such as *lines of code* and *hard-coded strings* consistently show strong correlation with defective scripts across datasets. In addition to our machine learning results being in agreement with the original results: Random Forest and Logistic Regression have a tendency to be the best predictive performers, with AUCs ranging from 0.70–0.80, while simpler learners like CART and KNN perform less well. Some differences were observed (e.g., Naive Bayes performing well on the WIK dataset), yet general patterns are in agreement with the original paper.

The biggest threats to our replication are from data availability and labeling. We relied on authors' released datasets rather than creating defect labels ourselves because it entails subjective judgment and large-scale commit analysis. Additionally, the migration of open-source projects away from Puppet reduces the external validity of direct application of the same approach today.

Despite these limitations, our replication helps by validating that interactions between source code attributes and defectiveness reported in the literature are reproducible, and confirming that property-based models are a viable option for IaC defect prediction. For future work, we suggest extending analysis to other modern IaC tools such as Ansible, Chef, or Terraform, updating datasets to reflect current practice, and exploring hybrid solutions combining source code features with process or text features. Such directions would make defect prediction research more relevant to the rapidly evolving DevOps landscape.

References

[1] M. Rahman and L. Williams, "Source Code Properties of Defective Infrastructure as Code Scripts," in *Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 246–256.

[2] GitHub REST API Documentation. Available: https://docs.github.com/en/rest

[3] Puppet Documentation. Available: https://puppet.com/docs/puppet/latest/puppet_index.html