Vrije Universiteit Brussel

On the practice of semantic versioning for Ansible galaxy roles: An empirical study and a change classification model

Opdebeeck, Ruben; Zerouali, Ahmed; Velázquez-Rodríguez, Camilo; Roover, Coen De

Published in:
Journal of Systems and Software

DOI:
10.1016/j.jss.2021.111059

Publication date:
2021

License:
CC BY-NC-ND

Document Version:
Accepted author manuscript

Link to publication

Citation for published version (APA):
Opdebeeck, R., Zerouali, A., Velázquez-Rodríguez, C., & Roover, C. D. (2021). On the practice of semantic versioning for Ansible galaxy roles: An empirical study and a change classification model. *Journal of Systems and Software*, *182*, [111059]. https://doi.org/10.1016/j.jss.2021.111059

Copyright
No part of this publication may be reproduced or transmitted in any form, without the prior written permission of the author(s) or other rights holders to whom publication rights have been transferred, unless permitted by a license attached to the publication (a Creative Commons license or other), or unless exceptions to copyright law apply.

Take down policy
If you believe that this document infringes your copyright or other rights, please contact openaccess@vub.be, with details of the nature of the infringement. We will investigate the claim and if justified, we will take the appropriate steps.

Download date: 23. Mar. 2025

On the Practice of Semantic Versioning for Ansible Galaxy Roles: An Empirical Study and a Change Classification Model

Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, Coen De Roover

Software Languages Lab – Vrije Universiteit Brussel
{*ropdebee, azeroual, cavelazq, cderoove*}*@vub.ac.be*

Abstract

Ansible, a popular Infrastructure-as-Code platform, provides reusable collections of tasks called roles. Roles are often contributed by third parties, and like general-purpose libraries, they evolve. Therefore, new releases of roles need to be tagged with version numbers, for which Ansible recommends adhering to the semantic versioning format. However, roles significantly differ from general-purpose libraries, and it is not yet known what constitutes a breaking change or the addition of a feature to a role. Consequently, this can cause confusion for clients of a role and new role contributors.

To alleviate this issue, we perform an empirical study on semantic versioning in Ansible roles to uncover the types of changes that trigger certain types of version bumps. Our dataset consists of over 81 000 version increments spanning upwards of 8 500 Ansible roles. We design a novel structural model for these roles, and implement a domain-specific structural change extraction algorithm to calculate structural difference metrics. Afterwards, we quantitatively investigate the state of semantic versioning in Ansible roles and identify the most commonly changed elements. Then, using the structural difference metrics, we train a Random Forest classifier to predict applicable version bumps for Ansible role releases. Finally, we confirm our empirical findings with a developer survey.

Our observations show that although most Ansible role developers follow the semantic versioning format, it appears that they do not always consistently follow the same rules when selecting the version bump to apply. Moreover, we find that the distinction between patch and minor increments is often unclear. Therefore, we use the gained insights to formulate a number of guidelines to apply semantic versioning on Ansible roles. These guidelines can be used by role developers to ensure a clear interpretation of the version increments.

Keywords: Ansible; Infrastructure as Code; Semantic Versioning; empirical study; mining software repositories

1. Introduction

Ansible is a popular Infrastructure-as-Code (IaC) tool for automatically deploying and configuring large-scale infrastructures. Ansible developers create playbooks containing a series of tasks, which can be automatically executed on a collection of hosts to obtain the desired infrastructure. These tasks may include installing a database driver, configuring a web server, etc. Such tasks can often be reused across playbooks, e.g., installing a database driver is often a similar process, regardless of usage context or platform.

To promote reuse and composition, Ansible offers roles, which, in their most basic form, are a series of reusable tasks. In addition, it hosts Galaxy[1], an online registry containing nearly 27 000 roles[2], contributed by third-party developers, which role clients can include into their playbooks. Roles often make extensive use of variables, so that its clients can parametrise its behaviour. For example, a role that installs a database driver could be parametrised to change the version of the installed database, specific configuration values for the driver, etc.

The "as Code" suffix in IaC does not merely signify that its files are written in textual form and interpreted by a machine. Instead, it encompasses every process surrounding regular source code, such as collaboration, version control, and importantly, evolution. Consequently, like general-purpose libraries, Ansible roles evolve over time, e.g., bugs get fixed, variables are added, tasks get refactored, etc. Thus, roles need to be versioned, so that new releases can be made available.

To denote role versions, Ansible recommends role developers to use the well-known Semantic Versioning (SemVer)[3] format (i.e., *major.minor.patch*). The SemVer specification states when each part of the version number ought to be incremented. Major version bumps are reserved for backwards-incompatible changes, whereas minor bumps should be applied when new features are added to the software's interface. Patch bumps should be carried out if the release does not change the interface, and contains only bug fixes, refactoring, etc.

1.1. Motivation

Function-based Application Programming Interfaces (APIs), such as those typically exposed by traditional software libraries, lend themselves well to the SemVer specifications. However, Ansible roles typically do not expose function-based APIs. Consequently, some of SemVer's rules may not readily apply to Ansible roles. Although Ansible recommends the SemVer *format*, it makes no mention of its rules, which may cause versioning practices to diverge among developers. For example, starting with Ansible 2.10, Ansible's maintainers began to move language extensions into community-maintained collections, which can group modules, plugins, roles, etc [1]. This change required imposing strict

[1] `https://galaxy.ansible.com`
[2] As of January 2021, `https://galaxy.ansible.com/search?type=role`
[3] `https://semver.org/`

adherence to SemVer on some collections, which led to some community maintainers having to adapt their versioning practices [2, 3]. Similarly, the primary goal of some roles is to install software, and thus its version may depend both on the role itself, as well as on the version of the software it installs. Some role developers have therefore started to use custom extensions to the SemVer format that include both [4].

Moreover, for traditional software libraries, previous research has shown it to be important to adhere to SemVer [5]. Breaking changes have a large impact on client software, and library clients are often hesitant to upgrade dependencies to new major versions. Anecdotal evidence suggests that role clients often suffer from breaking changes [6, 7]. One of the most prominent figures in the Ansible community even testifies suffering from breaking changes in upstream dependencies on a weekly basis [8]. Furthermore, improved support for role versioning has been a widely-requested feature for more than five years [9, 10]. Therefore, a loose interpretation of the SemVer specifications can be problematic. For instance, introducing breaking changes in minor version bumps [11, 12] may unexpectedly break downstream clients' playbooks [13]. Similarly, unnecessarily releasing major versions without breaking backwards compatibility [14] may lead to many downstream clients being hesitant to update. However, before one can properly investigate the extent of such issues in the Ansible Galaxy ecosystem, one must obtain a thorough understanding of the practice of role versioning.

1.2. Contributions

In this paper, we aim to uncover developer practices regarding the versioning of their Ansible roles. We are particularly interested in the changes that trigger a certain type of version increment. A better understanding of these changes would enable role clients to assess the effort needed before updating an installed role. Moreover, it can help role developers to align their role versioning with established practices, thereby making role versioning more consistent and reliable for their clients. Finally, it may provide researchers with a better foundation on which to build tool support for semantic versioning, or on which to conduct subsequent empirical studies into the impact of role changes.

There have been multiple studies investigating IaC (e.g., [15, 16, 17, 18]). However, these consider end users of IaC tools and snapshots of their files. To the best of our knowledge, we are the first to investigate a new side of IaC ecosystems, namely reusable roles, as well as the evolution of IaC files. More specifically, this paper makes the following contributions:

1. We design and implement a novel hierarchical model of Ansible role structure, and use it to develop a novel domain-specific structural change extraction algorithm.

2. We carry out a quantitative study investigating the use of SemVer in Ansible role development and which types of changes lead to a certain type of version bump.

3. We qualitatively investigate developer practices regarding SemVer by means of a developer survey.

4. We develop a classification model to predict the applicable SemVer version bump type using the distilled structural changes.

5. We evaluate this classification model through a longitudinal study.

6. We formulate a number of versioning guidelines based on the results of the developer survey and the features deemed most important by the classification model, and make a number of concrete recommendations on versioning of Ansible roles.

To conduct our study, we employ a dataset containing more than 8 500 roles, 90 000 role versions, and 81 000 version increments [19]. Our tools for analysis are available in a replication package found at `https://doi.org/10.5281/zenodo.4992072`.

This article is an extension of a previous conference paper [20]. In this journal extension, we present a new research question, RQ_4. Moreover, we perform more in-depth analyses into the pre-existing research questions, and use a larger dataset. As a result, the second and third contribution are expanded, and the fifth and sixth contribution consist solely of previously unpublished material.

2. Related work

In this section, we summarise a selection of research on Infrastructure as Code and Semantic Versioning, and highlight key differences with our work.

2.1. Infrastructure as Code

Infrastructure as Code is an emerging research domain, with an increasing number of works published each year [21]. Industrial IaC practitioners often face the challenge of identifying defects in their files [15]. As a result, a large proportion of existing work on IaC has focused on defect prediction and detection. One such topic is verifying semantic requirements of IaC files [22, 23, 24]. Other researchers have focused on syntactical properties, metrics, smells, and detection rules to highlight potential problems [25, 16, 26, 18, 17, 27]. For example, Rahman and Williams [25] constructed defect prediction models using 10 source code properties. Sharma et al. [26] proposed a catalogue of 24 design and implementation smells for IaC code. Van der Bent et al. [17] defined and empirically validated a suite of maintainability metrics for Puppet code. Building upon this, Dalla Palma et al. [18] proposed a suite of 46 metrics for Ansible.

The aforementioned studies focus on defects in IaC files created by end users of IaC tools. Moreover, they only consider snapshots of such IaC files, and mostly remain on a syntactical level. On the contrary, we focus on reusable IaC

files created by community contributors, an understudied part of the IaC ecosystem. Specifically, we look at the evolution of such files, rather than snapshots. Instead of focusing on the syntactical level, we mainly consider the structure of these reusable IaC files, making our approach less sensitive to coding styles.

2.2. Semantic Versioning

The semantic versioning specifications are commonly recommended by package managers to denote the kind of changes in new releases of software libraries. Because of its importance, SemVer has been subjected to many research studies. Raemaekers et al. [5] investigated the usage of SemVer in Java libraries on Maven Central over a seven-year period. They found that library maintainers did not respect SemVer (e.g., a third of minor releases introduced a breaking change), and that the adherence to SemVer increases only marginally over time.

Bogart et al. [28] performed case studies on three software ecosystems (i.e., *Eclipse*, *CRAN*, and *npm*) to understand how developers make decisions about changes and their costs. They found that the three ecosystems differ significantly in their practices and policies. The same researchers conducted a survey about common practices among over 2 000 developers in 18 ecosystems [29]. They observed that maintainers generally try not to perform a breaking change, with most developers across all ecosystems reporting less than one breaking change a year. Maintainers commonly bundle multiple breaking changes together to avoid disruptions. Finally, they observed that the frequency of breaking changes is higher in some ecosystems (*npm*, *Rust*) than others (*Perl*, *CRAN*, *Eclipse*).

Decan et al. [30] empirically studied SemVer compliance in four ecosystems (*Cargo*, *npm*, *Packagist*, and *Rubygems*) by analysing package dependency constraints. They found that the proportion of compliant dependency constraints increases over time in all ecosystems, and identified factors that influence the degree of compliance. Similarly, Dietrich et al. [31] studied over 70M dependencies in 17 package manager ecosystems, found that many ecosystems support flexible versioning practices, and that the adoption of SemVer is increasing in some.

3. Ansible Primer

Ansible is a popular infrastructure-as-code platform used to automate the deployment and configuration of multi-machine infrastructures. Although it is mainly intended as a tool to quickly set up a group of remote machines called *hosts*, it can equally be used to set up a local machine, e.g., for on-boarding. Ansible uses YAML as a domain-specific language for its infrastructure configuration files. Thus, most of its concepts are defined as key-value pairs in YAML files. There are various concepts in Ansible, e.g., inventories and plugins, which are outside of the scope of this primer. Furthermore, many of Ansible's elements accept a vast range of keywords, most of which will be omitted from this primer. Instead, we focus on two core concepts, namely the *playbook*, containing the infrastructure definition, and *roles*, which are reusable Ansible components, frequently contributed by third-party developers.

```
1  - hosts: localhost
2    vars:
3      base_name: "main"
4      remove_log_file: no
5    tasks:
6      - name: Compile the LaTeX file.
7        command: "pdflatex {{ base_name }}.tex"
8      - name: Remove log file, if enabled.
9        file:
10         path: "{{ base_name }}.log"
11         state: absent
12       when: remove_temp_files | bool
```

Listing 1: A contrived example of an Ansible playbook.

3.1. Playbooks

Ansible's flagship concept is the *playbook*, a central definition of the process of deploying and configuring complex infrastructures on a group of machines. Listing 1 depicts a playbook that compiles LaTeX files. Playbooks contain *plays*, each describing the configuration of a group of hosts. The example defines one play, targeting the local machine (line 1). Playbooks may define multiple plays, e.g., one to configure database servers, and another to set up a load balancer.

Each play has its own set of *variables*, defined as key-value pairs (lines 2–4). Variables can be used in template expressions, enclosed by double braces (e.g., line 7), which are evaluated lazily. These variables can be used inside of the play's *tasks*, of which our example defines two (lines 6–7, 8–12). Tasks are executed in sequential order, and each task executes a single *action*. For example, the first task executes `command`, which runs the `pdflatex` program (line 6). The second task uses the `file` action to ensure a file is absent from the file system. The path to the file, and the desired state, are given as the action's arguments (lines 10–11). Tasks can also be executed conditionally by specifying a condition using the `when` keyword (line 12). Other keywords exist to adjust the control-flow semantics of a task, such as `loop`, which iteratively executes a task for each item in a list.

Besides plays, variables, and tasks, Ansible offers two more important concepts. *Blocks* can be used to group tasks or other, nested blocks, and also offer exception handling mechanisms. A *handler* is a special type of task that can be used to react to changes made by a task. A task can notify a handler, which registers the handler to be executed at the end of the play. If not notified, a handler is not executed.

3.2. Roles

Ansible provides *roles*, an abstraction for multiple reusable IaC files containing tasks, variables, handlers, etc. Although similar to plays, roles ought to be generic to be reused across playbooks. When a role is imported into a play, the tasks, handlers, and variables defined in its files are embedded into the play as if they were part of the play itself. Since roles consist of multiple files, they follow a strict directory structure, with subdirectories for each element type, as follows:

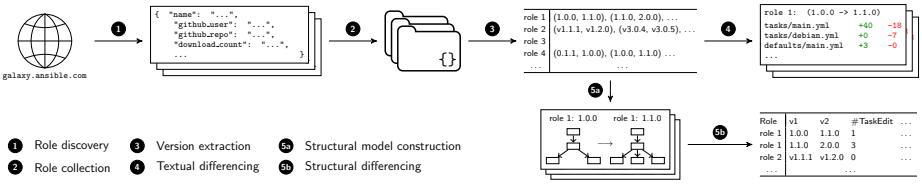


Figure 1: Overview of the data extraction pipeline.

- Files in the `tasks` directory contain the role's blocks and tasks, whereas the `handlers` directory contains the role's handlers. These directories must contain a `main.yml` file, which Ansible loads by default. Other files may be present, but must be imported manually through a task.

- Files in the `defaults` and `vars` directories contain default variables and role variables, respectively. The difference between these variables is their precedence. Role variables are difficult to override as a client, and are therefore often used as constants. Default variables are in contrast much easier to override, and serve as the means for the client to parametrise the role's behaviour. Similarly to tasks and handlers, the `main.yml` files in these directories are loaded by default, while other files may contain additional variables which can be loaded manually.

- The `meta/main.yml` file contains metadata for the role, such as author, description, license, etc. This also lists the platforms with which the role is compatible, and any other roles on which this role depends. These dependencies are executed before the role itself is.

- The `files` and `templates` directories contain resources for the role to use, such as configuration files. Files in the latter can be parametrised by the role's client using variables.

To ease the discovery of third-party roles, Ansible provides Galaxy, a central registry which, as of January 2021, contains over 26 500 open-source, reusable roles provided by the community. Since roles can evolve over time, they should also be versioned, and Galaxy provides version information such that role clients can install specific role versions. To import versions, Galaxy scans the role's git repository for tags matching the SemVer format. It thereby recommends using the SemVer format to denote versions, however, this is not a strict requirement. Moreover, this applies only to the format, and Ansible does not provide guidelines stating when each type of bump should be applied.

4. Method and Data Extraction

To investigate the versioning of Ansible roles, we make use of a dataset containing more than 26 500 unique Galaxy roles written by over 6 000 unique authors [19]. Of these, nearly 9 000 roles have more than one version, making them suitable for our investigation. This section describes the pipeline used to

Table 1: Overview of the collection and filtering of the dataset. The bottom row, highlighted in bold, depicts the final count of roles and versions considered for analysis.

Stage	#Roles	#Authors	#Versions	#Incr.
1. Role discovery	26 834	6 478	N/A	N/A
2. Role collection	25 390	6 081	N/A	N/A
3a. Tag extraction	12 326	2 737	101 488	N/A
3b. Version extraction	11 156	2 432	93 960	82 796
3c. Version bump extraction	8 560	2 053	90 051	81 253

collect this data and extract the necessary information. Figure 1 depicts an overview of this pipeline, while Table 1 summarises the dataset.

The pipeline consists of 5 stages. We first discover the roles from Galaxy, clone their repositories, and extract their versions. We extract line-based textual difference metrics, i.e., the number of lines inserted and deleted, for each version bump. However, textual differences are sensitive to semantically-irrelevant implementation details such as refactoring, comments, and whitespace. To combat such problems, we also look at the differences between role versions on a structural level using a novel structural model and specialised tree differences, described in Section 5. We calculate metrics of difference, rather than difference in metrics, for increased accuracy. For example, although comparing the number of tasks in two versions would reveal a net increase or decrease, it would not reveal that a task was added while another was removed.

4.1. Role Discovery

The first stage of our pipeline discovers open-source roles from the ecosystem using the Galaxy role registry. We notice that Galaxy may sometimes lack some role versions, and may list roles which were erroneously imported (e.g., named "test"). Therefore, we do not rely on Galaxy metadata to extract role versions, and will subsequently clone the role repositories instead. Ultimately, we extracted 26 834 roles written by 6 478 unique authors. However, as we will see shortly, many of these roles contain no versions, are thus irrelevant for our study, and are therefore removed in subsequent pipeline stages.

4.2. Role Collection

In the second stage, we clone the git repositories of each collected role through the GitHub URL obtained from Galaxy. We successfully cloned the repositories of 25 639 roles discovered in the previous phase. The remaining 1 195 repositories could not be cloned, as they were not available. These repositories may have been removed or made private since they were added to Galaxy. We excluded 249 roles which are part of a repository that is linked to multiple roles in the dataset. Such "monorepos" have a single version shared among all of its roles. Consequently, a change to one role in such a repository would impact the version of all other, unchanged roles as well. This reduces the size of our dataset to 25 390 roles, contributed by 6 081 authors. Of these, 8 289 roles have

been in active development in the previous year, i.e., have at least one commit in the 12 months prior to collection.

4.3. Version Extraction

We then move on to extracting role versions from the cloned repositories' tags. We extract each tag from every role repository, leading to 101 488 different tags across 12 326 roles. The remainder of the roles did not contain any tags. We then perform further filtering on these tags to retain only consecutive semantic version bumps.

First, we remove all tags that do not match the SemVer *major.minor.patch* format. Note that we also remove pre-release versions (e.g., `1.0.0-alpha`), since pre-release versions might not follow all of the SemVer specifications. Therefore, we focus solely on "main" releases. This filtering reduces the number of tags in our dataset to 93 960, belonging to 11 156 roles written by 2 432 authors.

Subsequently, we construct version bumps from the consecutive semantic versions for each role. Any role with less than two versions is omitted, since they are of no interest to our study. Moreover, we exclude version increments that are not strictly consecutive. To illustrate, consider three consecutive tags such as `2.2.4`, `2.2.4.1`, and `2.2.5`. The second of these tags does not match the SemVer format, and thus, neither the increment of `2.2.4` to `2.2.4.1`, nor `2.2.4.1` to `2.2.5` is included in the dataset because of earlier filtering. Moreover, we do not include an increment for `2.2.4` to `2.2.5`, since they are not strictly consecutive. Although one could argue that this constitutes a patch bump, given the abundance of increments and the temporary lapse in adherence to SemVer, we decided to conservatively exclude such bumps.

After this filtering, our dataset consists of 8 560 roles written by 2 053 authors. In total, it contains 90 051 versions and 81 253 version increments.

4.4. Textual Differencing

Following the third stage, our pipeline branches into two strategies to extract difference metrics between successive versions. The first of these, marked as stage 4 in Figure 1, extracts line-based textual difference metrics and commits between two role versions. The line-based differences are obtained using the `git diff` command, from which we extract which files were added, deleted, moved, or edited, as well as the number of lines inserted and deleted in each file. Additionally, we extract these features between an empty repository and the role's first release, to estimate the amount of effort needed to create the initial release of a role.

4.5. Structural Differencing

The second strategy to extract difference metrics uses our structural model and differencing algorithm (Section 5). We extract a structural model from the source code of each role version according to the versions gathered in stage 3 (stage 5a). For each version increment, the pair of structural models is fed into our domain-specific structural differencer (stage 5b), which produces a sequence

of fine-grained differences belonging to one of 41 different change types. This led to 719 424 individual distilled changes. Then, we produce structural difference metrics by counting the number of occurrences of each change type for each role version increment.

4.6. Difference to the Previous Dataset

In our previous work, which this paper extends, we used an older version of this dataset extracted on June 12th, 2020 [20]. In this paper, we are using a more recent version of the dataset, gathered on January 20th, 2021 [19]. Since Ansible Galaxy is an up and coming ecosystem, this period of 6 months has caused a relatively large growth in the dataset. In this subsection, we summarise the most important changes between both datasets.

One can immediately observe a net increase in the number of entities in the dataset. More than 2 000 roles and nearly 500 authors have been added to the dataset. In fact, the new dataset contains 2 523 roles that were not present in the old dataset. However, this number is likely a slight overestimation, as we matched roles across the two datasets based on their namespace and name. If any of these two were renamed between the time points at which the datasets were collected, the role would be considered new. It should be noted that not all of these roles are necessarily created after the previous dataset was collected. Instead, they may have existed prior to June 12th, 2020, but not yet have been published to Galaxy.

Out of these 2 523 new roles, 1 093 contain more than one semantic version. This led to a total of 7 924 version increments being added to the dataset. Another 7 192 new version increments come from new versions released for 1 829 pre-existing roles. Together, these lead to more than 15 000 new version increments that have not been investigated previously, an increase of more than 20% compared to the old dataset.

317 roles in the old dataset could not be matched to roles in the new dataset. These roles have either been removed from Galaxy, or renamed. Naturally, any versions associated with these roles are absent in the new dataset as well.

Finally, one can observe that structural change distilling is applied to many more version increments than in our previous work. Previously, we only distilled such changes for roughly 89% of the version increments, whereas now, they are extracted for all increments. In previous work, our structural model extractor would reject any role version which contained syntactical errors, whereas now, the extractor attempts to parse as much code as possible. This enabled it to parse all role versions, and consequently, we could distil structural changes for all version increments.

5. Structural Model Differencing

This section describes our novel structural model for Ansible roles, as well as the accompanying structural differencing algorithm. This model and algorithm are used in the final stage of our pipeline, described in Section 4, to extract metrics of the difference between two role versions.

5.1. Structural Model

Our novel structural representation of Ansible code is a tree of Ansible elements, such as blocks, tasks, and variables. It is inspired by, and extracted from, Ansible's own internal representation of its files. This internal representation is structural in nature and therefore transcends the syntactical contents of files, but this comes at the cost of having to undo some of the optimisations made by Ansible that render executing the IaC files more efficient. For example, when Ansible encounters a task which statically imports another task file, it replaces the importing task by the content of this file. For dynamically included files, this is impossible, since the file's name may depend on variables which are only known at run time. Since we aim for a structural representation, it is important to represent all of the task files separately, rather than inlined into another file.

Figure 2 depicts our structural model. It is a hierarchical tree of four main types of elements, namely files, variables, blocks, and tasks, closely following the structure of a role, as described in Section 3.2. The first level of the hierarchy consists of a series of files, each type representing a file in either the `tasks`, `defaults`, `vars`, `handlers`, or `meta` directory. There may exist multiple files of each type, distinguished by their name, except for metadata files.

The metadata file is a singleton, and always represents the `meta/main.yml` file, if present. Our representation of a metadata file contains exactly one meta-block, whose attributes represent the file's contents, e.g., platforms and dependencies.

Files in `tasks` and `handlers`, as well as their contents, are represented near-identically, although we make a clear distinction to represent the difference in control-flow semantics (cfr. Section 3.1). Such files are internal nodes whose children are blocks. Blocks, in turn, have their contents, i.e., nested blocks and tasks, as children. In both cases, the children are ordered by their execution order. Top-level tasks, i.e., tasks not contained inside of a block, are placed in an implicit block for uniform representation. For blocks and tasks, we additionally store their key-value pairs as attributes.

Files in `vars` and `defaults` are represented as role variable files and default variable files, respectively. Again, the two node types and their children are similar, yet we make the distinction to enforce the differences in precedence (cfr. Section 3.2). Each of these files is an internal node of our hierarchy. Their children are leaf nodes representing the variables in the file, which store the name and assigned value. Contrary to tasks and handlers, we do not define an order for these children, since their order is semantically irrelevant because of lazy loading.

To create the structural model, we employ Ansible's internal parser and post-process its representation. Thus, we benefit from the parser's ability to transform different syntactical styles into the same constructs. This also ensures we do not attempt to include an ill-formatted file into the structural model. Instead, these are ignored.

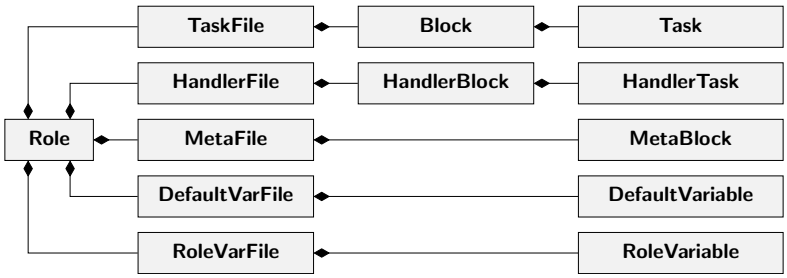


Figure 2: The condensed UML class model of the structural role model.

5.2. Structural Differencing

Our structural differencing algorithm extracts a set of fine-grained changes that represent the difference between a pair of structural models. Rather than relying on generic tree differencing algorithms, such as ChangeDistiller [32], GumTree [33], or ChangeNodes [34], we designed a domain-specific one. Whereas the aforementioned algorithms identify changes between abstract syntax trees, we use structural trees, which are conceptually different. Moreover, we aim to extract specific changes that can serve as a detailed summary of a role release, which is not possible with a generic approach. Finally, implementing our own differencer enables us to apply domain-specific knowledge to increase the usability of the extracted changes, by disregarding changes carrying no semantic relevance.

5.2.1. Change Types

Our domain-specific algorithm can produce 41 different change categories, constructed by combining 4 main change types and the element types of the structural model. The valid combinations of change types and element types are depicted in Table 2. The table's columns represent the 4 main change types. *Additions* and *removals* represent changes where a node has been added or removed in the second tree. *Relocations* represent changes where a node was moved to a new position, either in the same parent (local relocation), or to a new parent (global relocation). Finally, *edits* represent changes where a node's value was edited. Note that each change applies to an individual node, e.g., when an internal node is added, additions for its children are distilled individually, and when an internal node is relocated, its children are not relocated individually, since they retain the same position in the same parent.

Table 2's rows contain various element types, and non-empty cells mark possible combinations of change type and element type. All non-root node types of the model are present as rows, except for metadata-related nodes, since these are singletons. Instead, we represent additions and removals to the platform and dependency sets separately, and consider any other change to the metadata as a generic edit. The other main change types are not applicable to the metadata. Edits to the four remaining file types are represented by edits to its individual children, and a relocation of a file means that its name has changed. We make no attempt to identify renames of variables, instead approximating such cases

Table 2: The valid combinations of orthogonal change categories.

	Addition	Removal	Edit	Relocation
Dependencies	✓	✓	-	-
Platforms	✓	✓	-	-
Misc. metadata	-	-	✓	-
Default var.	✓	✓	✓	✓
Role var.	✓	✓	✓	✓
Default var. file	✓	✓	-	✓
Role var. file	✓	✓	-	✓
Task	✓	✓	✓	✓
Handler task	✓	✓	✓	✓
Block	✓	✓	✓	✓
Handler block	✓	✓	✓	✓
Task file	✓	✓	-	✓
Handler file	✓	✓	-	✓

as an addition and a removal, since for a client of the role, both a rename and a removal are potentially breaking changes. Therefore, an edit to a variable indicates that its assigned value has changed, but its name remained unchanged.

5.2.2. Change Distilling

We will now describe our domain-specific structural change distilling algorithm. The main challenge in extracting changes is identifying relocations of edited elements, e.g., a task that was moved to another block while simultaneously having its keywords edited. We also prefer extracting relocations rather than a pair of addition and removal, since the latter would over-approximate the number of elements added and removed in the new tree. While extracting relocations, we prioritise relocations to positions that are intuitively "close" to the old position, which leads to more intuitive changes.

At a high level, our algorithm is similar to ChangeDistiller [32], and works as follows. We compare the two given structural models in a depth-first manner. For each internal node, we compare its children and extract additions, removals, and edits. We identify edited nodes by calculating their similarity, which is a number between 0 and 1 inclusive, where larger values mean more similarity, and apply a threshold of 0.5, essentially meaning that if two nodes are not at least 50% similar, the change is represented as an addition and removal instead. After these changes have been identified, we look for local relocations in the direct children of the internal node by matching nodes removed from the old subtree to nodes added in the new subtree, again using a 0.5 similarity threshold. Finally, we check for global relocations in a similar way, but now also consider additions and removals of indirect children.

The major difference between our algorithm and ChangeDistiller [32]

is that ours is specific for our structural model, which enables us to use domain knowledge to improve its results. For example, flat sequences such as role dependencies and compatible platforms can be compared more easily without having to use tree traversals. Moreover, the order of variable definitions makes no difference, and thus, we can sort them by name to speed up the extraction of edits, and do not have to look for relocations.

The main benefit of the domain-specificity is that it enables us to define highly-specialised similarity metrics for each element type. For instance, we define the similarity of files containing variables as the proportion of variables that are common between both files, additionally incurring a penalty for each variable whose assigned value differs. This penalty allows us to distinguish between two files defining the same set of variables with distinct values. For tasks, the similarity is calculated in terms of the number of common keywords with the same value. As a final example of a specialised similarity metric, for blocks, the similarity is calculated as the average of pair-wise similarities of its contents, with an additional penalty for each child that would be locally relocated. Again, this penalty allows us to distinguish two blocks that execute the same set of tasks, but in a different order.

6. Empirical Analysis Results

The research questions in this study are organized into three parts: 1) a quantitative analysis that includes four research questions RQ_0 to RQ_3; 2) a qualitative study where we conduct surveys with Ansible role developers; and 3) the building of a classification model to predict the suitable SemVer version increment for a new role release.

To confirm the observations of our empirical analysis, we carried out comparisons of statistical distributions using the *Mann-Whitney U* test, a non-parametric test where the null hypothesis H_0 checks if two distributions are identical without assuming them to follow a normal distribution, the alternative hypothesis H_1 being that one distribution is stochastically greater than the other. For all statistical tests in the paper considered together, we wish to achieve a global confidence level of 95%, corresponding to a value of $\alpha = 0.05$. To achieve this overall confidence, the p-value of each individual test is compared against a lower α value, following a Bonferroni correction. If n different tests are carried out over the same dataset, for each individual test one can only reject H_0 if $p < \frac{0.05}{n}$. In our case $n = 12$, i.e., $p < 0.004$.

To report the *effect size* of the statistical tests, we use *Cliff's Delta* d, a non-parametric measure that quantifies the difference between two populations beyond the interpretation of p-values. Using the thresholds provided by Romano et al. [35], we interpret the effect size to be *negligible* if $|d| \in [0, 0.147[$, *small* if $|d| \in [0.147, 0.33[$, *medium* if $|d| \in [0.33, 0.474[$ and *large* if $|d| \in [0.474, 1]$.

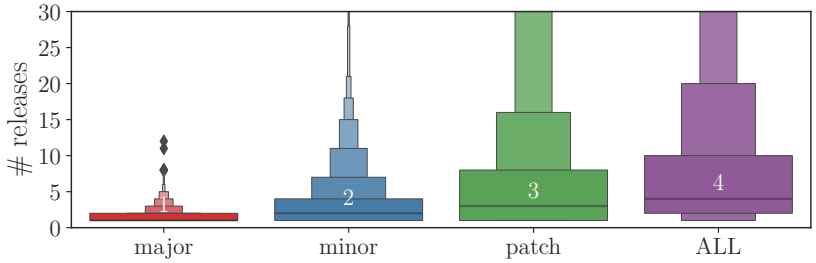


Figure 3: Distribution of the number of Ansible role releases per release type, i.e., patch, minor and major, with the median value annotated in white.

6.1. Quantitative Analysis

RQ_0: How many Ansible roles use the SemVer format?

Before we can investigate the meaning of SemVer in Ansible roles, we need to verify that their versions syntactically adhere to the SemVer format, i.e., *major.minor.patch*. We thus checked whether the role versions in our dataset (cfr. Section 4) match this version. We found that 91.1% (100 999) of the version numbers use this format[4]. The remaining, non-compliant version numbers belong to 2 372 roles, of which 1 177 roles have only non-compliant versions. Moreover, 11.2% of all developers that added tags to their repositories (306) only use non-compliant versions. All such non-compliant versions are removed from subsequent processing.

We also checked whether there exist version numbers with additional labels, like pre-releases (e.g., `1.0.0-alpha`). We found that only 0.65% (611) of the versions contain such labels, belonging to 225 roles. Therefore, we decided to remove these pre-release versions, since there are only a small number of them in the dataset. Moreover, pre-releases may be unstable and not comply with compatibility requirements[5], and thus the changes may not be representative.

Next, focusing only on those versions that are complying with the SemVer format, we identify their type of version bump, i.e., for every two successive versions we check which version number changed. For example, an increment of `2.1.0` to `2.1.1` is considered a *patch* release, whereas an increment of `2.0.z` to `2.1.1` is considered a *minor*. Similarly, an increment of `1.y.z` to `2.1.1` would be considered a *major* release. We found that 12.1% of all releases are initial releases (i.e., the first release of each role), 63.5% are patches, 20.1% are minors and 4.3% are major releases. Figure 3 depicts boxen plots that show the distribution of the number of each release type per Ansible role. It can be seen that roles release patches more often than minors, which are in turn more common than majors. The median number of patch, minor, and major releases

[4]Note that this count includes pre-release versions, therefore making it higher than the number of versions extracted in Section 4.

[5]`https://semver.org/#spec-item-9`

per role are 3, 2, and 1, respectively. *Mann-Whitney U* tests between the three pairs of distributions confirmed that there is a statistical difference for each pair. These results are to be expected when software adheres to the SemVer specifications. Patch releases contain bug fixes, often frequent and easier to create, whereas adding new functionality in minor releases takes more effort and thus happens less often. Major releases contain breaking changes, which ought to be rare.

Figure 3 also shows two roles having 11 and 12 major releases. A manual inspection of these two roles revealed that in the first case[6], major releases are appropriate, i.e., whenever the version of the underlying software is upgraded, a new major version is released, whereas for the second case, major releases do not follow any specific scheme. In fact, we looked at other roles created by the author of the second role and we found that many major versions are released without any breaking change.

When considering all releases and disregarding their types, we found that the median number of releases per role is 4. We also found that 23 roles are releasing more often than other roles, having more than 100 versions since their creation. A deeper investigation revealed that these roles have a median of 3 commits per release. We manually looked at these roles and we found that 10 of them are maintained by the same developer. Nevertheless, in most of these cases, the changes are arguably too small to deserve their own release, and could have been grouped into a larger one instead. As Ansible Galaxy only supports installing roles using version tags, it might be possible that the developers of these roles wanted the clients that use Galaxy to benefit from the most recent changes even if they are not so important.

Findings: The majority of Ansible roles use the SemVer format. Patch releases are more common than minors, while minors are more common than majors.

RQ_1: How much effort does it take to create role versions?

RQ_0 shows that patch and minor bumps are more common than major bumps. Building upon this, in this research question we investigate how much effort it takes to create each type of release, i.e., the number of commits and changes that are needed to release new versions of Ansible roles.

Figure 4 depicts the distribution of the number of commits used to create the initial role release and a patch, minor, and major release, from any immediately preceding release. We observe that major releases generally require more commits than minors, which in turn require more commits than patches, which require only a small number of commits. Initial role releases show higher outliers, indicating that the very first role may require more commits than any other release. To statistically confirm our observation, we carried out *Mann-Whitney U* tests between all pairs of distributions. For each comparison, we

[6] `https://github.com/githubixx/ansible-role-kubectl`

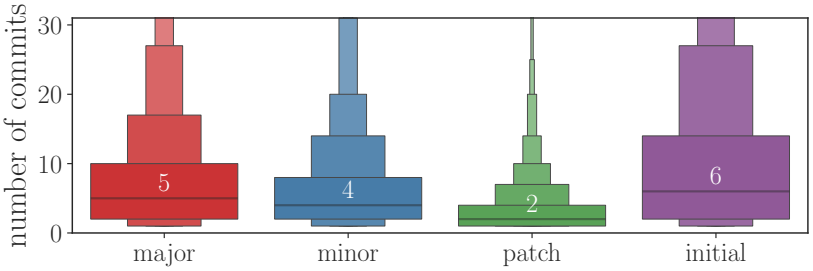


Figure 4: Distribution of the number of commits for each type of version transition, with the median value annotated in white.

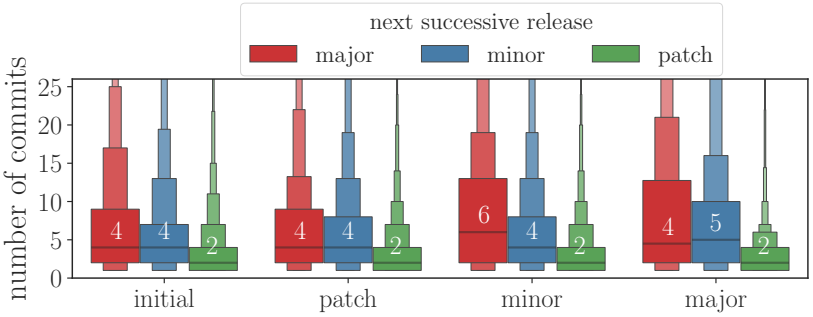


Figure 5: Distribution of the number of commits used to release initial, patch, minor and major version, with the median value annotated in white.

found a statistical difference. However, we found that the effect size between minor and major distributions is *negligible* with $|d| = 0.06$. The mean and median number of commits are 11.6 and 5 for majors, 8.4 and 4 for minors, 3.7 and 2 for patches, and 15.7 and 6 for the initial release, respectively. We additionally investigated the number of commits for each type of version transition (e.g., initial to patch, major to minor, etc.) but could not observe any significant differences (see Figure 5). This suggests that the required effort depends on the target version rather than the former version.

Since the size of the changes in a commit may vary, we additionally analyse line-based difference metrics for each role release type. The distribution of the number of lines changed (i.e., insertions + deletions) for each type of release is depicted in Figure 6. The figure clearly shows that initial versions require significantly more line changes than any other release. This is to be expected, since the number of lines changed for initial releases coincides with the total number of lines in the role itself. We further observe that there are more lines changed in major releases than minors, and that patch releases require the least number of line changes. To statistically confirm our observation, we carried out *Mann-Whitney U* tests between all pairs of distributions. For each comparison,

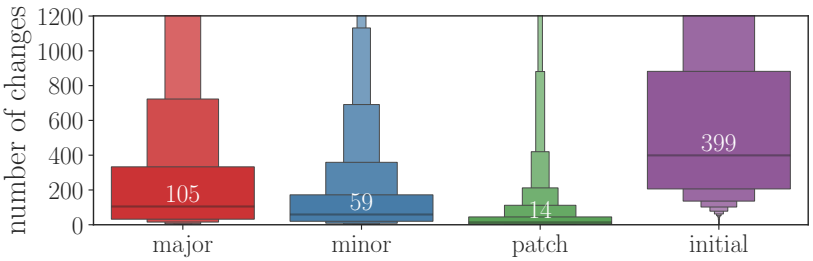


Figure 6: Distribution of the number of lines changed (insertions + deletions) in initial, patch, minor and major version bumps, with the median value annotated in white.

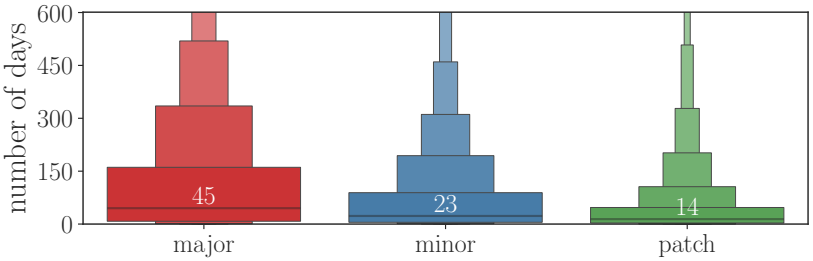


Figure 7: Distribution of the number of days required to create a major, minor or patch release from any other directly preceding release, with the median value annotated in white.

we found a statistical difference. However, we found that the effect size between minor and major distributions is *small* with $|d| = 0.19$.

Our observations are in line with the SemVer specifications. Major releases include breaking changes and therefore require more maintenance to the whole role. The addition of functionality in minor bumps requires more changes than a bug fix in a patch release.

Similarly, we computed the number of days between two successive versions. Figure 7 shows the distribution of the number of days required to create a patch, minor, and major release, from any immediately preceding release. We observe the same trend, in the sense that a major version requires more time than a minor version before it is released, while a patch release requires the least number of days. The mean and median number of days are 136.4 and 45 for majors, 82.8 and 23 for minors and 53.3 and 14 for patches, respectively.

Findings: Releasing new major versions of Ansible roles requires more commits, time and changes than minors. Patches require the least amount of effort of any release type.

RQ_2: Which role directories are touched between two role releases?

As shown in Section 3, Ansible roles have a specific structure. In this research question, we investigate, for each release type, which changes are most commonly applied to a role. We first look into line-based changes for each directory, after which we look into detailed structural differences.

First, without differentiating between role release types, we identify which directories have been textually edited most often. We found that only 7 of the role directories are touched in more than 10% of the releases. The top changed directories and the proportion of releases that changed them can be found in Table 3. We observe that tasks are the most commonly textually edited element in all release types, followed by default variables. We also notice two non-standard directories `tests` and `molecule` that have been touched in more than 10% of releases, i.e., 13.9% and 14.3%, respectively. These directories are used for testing. We observe that most directories are changed more often in major releases than any other releases. We also notice that the tests directory is touched more often than the vars directory when roles are releasing new major versions.

To obtain more insight into the number of changes that happen to the files in these directories during each release, we have computed the number of lines changed (i.e., insertions + deletions) for each file existing in these directories. Figure 8 shows the distribution of the number of changes that happen to the most commonly touched directories, grouped by release type. We can clearly observe that all directories have the least number of changes during patch releases. We also observe that the highest number of changes happen in major releases, except for the testing directory `tests` where we notice that more changes happen during minor releases.

Table 3: Proportion of releases that changed files in the main role directories, grouped by release type.

Directory	Patch	Minor	Major	All
`tasks`	47.8	72.4	78.9	55.0
`defaults`	29.0	56.2	67.5	37.1
`meta`	22.9	33.7	54.1	26.9
`templates`	15.9	32.7	37.9	20.8
`vars`	12.8	23.7	28.6	16.1
`molecule`	13.1	16.6	19.7	14.3
`tests`	9.7	23.4	31.0	13.9
`handlers`	4.6	11.9	17.7	6.9
`files`	2.0	4.6	7.5	2.9

Next, we only focus on the top five main directories and investigate whether they are changed in isolation or are touched simultaneously with others. Figure 9 depicts Venn diagrams presenting proportions of releases that touched the main Ansible role directories. As can be seen in Figure 9a, the majority of releases touch multiple directories, suggesting that these directories are closely related

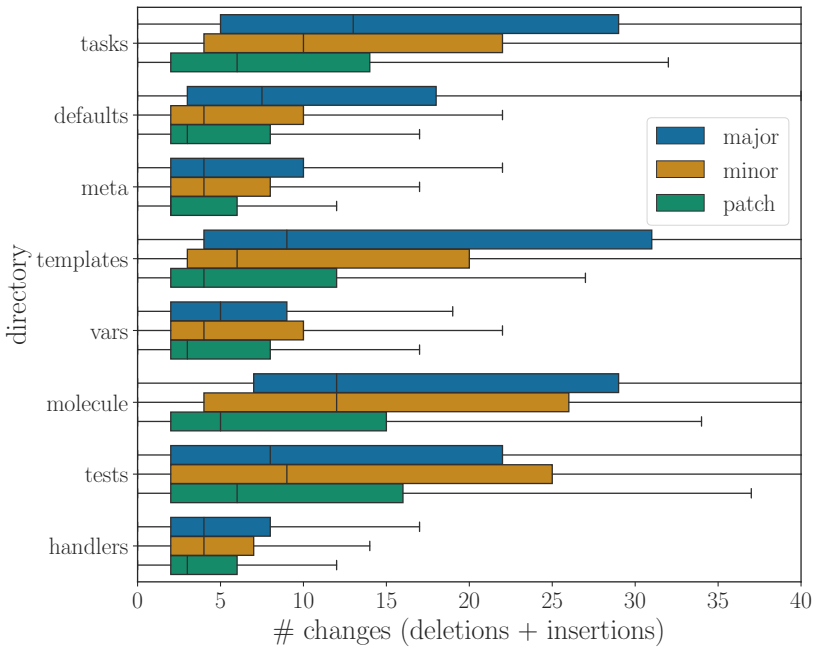


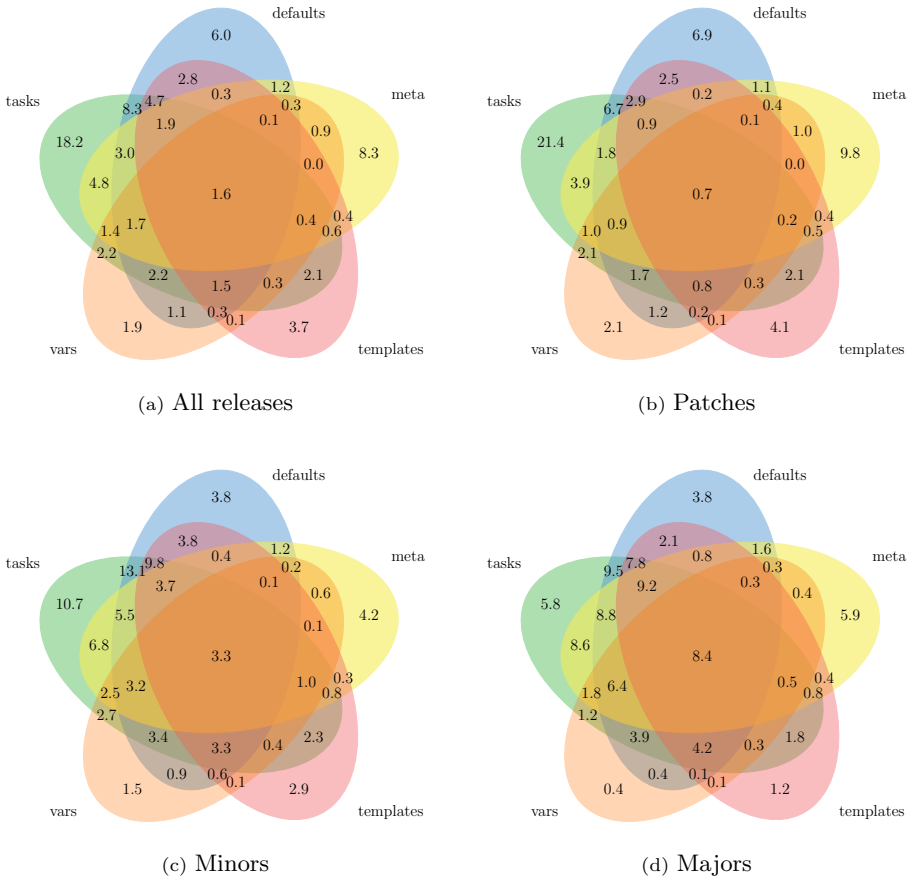
Figure 8: Distribution of the number of changes that happen to the most commonly touched directories, grouped by release type.

and changes to one directory may trigger changes to others. Figure 9d shows that major releases more often touch multiple directories, whereas Figure 9b shows that patch releases more often touch a single directory in isolation. This confirms that patches are mostly about small changes, such as bug fixes, whereas breaking changes in major releases require maintenance across the role, as was observed in RQ_1.

Findings: Major releases more often change multiple role directories, whereas patches more often change a single directory. Tasks and default variables are changed most often.

RQ_3: Which changes are made between two role releases?

Up to now, we have solely considered textual differences, which may include semantically-irrelevant changes such as refactorings. For this research question, we identify the role element types that are changed most often in specific bump types, using the changes extracted by our structural differencing algorithm (cfr. Section 5). Table 4 lists structural element types that have been changed (added, removed, relocated, or edited) in more role releases, along with the proportion of releases that changed such elements. We observe that all element types are more frequently changed in major bumps. Changes to tasks occur most frequently, in more than two thirds of the major releases and nearly half of



(a) All releases (b) Patches

(c) Minors (d) Majors

Figure 9: Proportion of Ansible role releases that changed files in combinations of the most commonly touched directories, grouped by release type.

all releases, whereas default variables are the second most frequently changed element. Note that the numbers in this table are all lower than the numbers provided in Table 3, showing that there exist releases which textually change those elements without incurring structural changes. We also looked further into these changes and found that the most commonly performed changes for tasks are edits (40.8% of all releases) and additions (22.89%), whereas for defaults, they are additions (20.25%) and then value edits (17.35%). Finally, for every release type, tasks are more often relocated or removed than variables. We also found that 25.93% of releases perform no structural change at all. This can be attributed to releases containing purely syntactical refactorings, or releases that perform changes in directories that are not considered in the structural model (e.g., files and templates, tests, etc.). Extra analysis also showed that regardless of the type of change, major releases perform more changes than minors or patches. Similarly, patches only perform a small number of changes.

This is in line with previous observations in RQ_2.

Table 4: Proportion of releases that changed a specific type of element, grouped by release type.

	Patch	Minor	Major	All
Task	42.76	67.53	74.49	50
DefaultVariable	26.25	52.46	63.85	34.11
MetaEdit	12.16	20.42	37.50	15.31
Block	9.06	27.45	38.59	14.73
RoleVariable	11.29	21.67	26.15	14.4
Platform	8.76	16.00	26.50	11.3
TaskFile	4.98	17.56	24.23	8.82
RoleVariableFile	4.13	9.58	11.62	5.75
HandlerTask	3.47	9.71	14.41	5.44
HandlerBlock	1.08	3.66	6.01	1.91
Dependency	1.27	2.82	6.99	1.91
DefaultVariableFile	1.03	1.58	1.97	1.2
HandlerFile	0.50	1.93	4.01	1

Findings: Textual changes often incur no structural changes. On a structural level, the most commonly applied changes are additions and edits of tasks and default variables.

6.2. Qualitative Study

To relate our findings to role versioning in practice, we contacted a number of popular Ansible role authors. We selected developers who have at least one role that has multiple versions and who have at least 2 500 role downloads in total, indicating reasonable popularity. We retained only those developers who had an e-mail address listed publicly on either their Ansible Galaxy profile or their GitHub profile. We contacted these developers via e-mail with the following questions.

1. Semantic versioning has traditionally been applied to software libraries. Do you consider an Ansible role to be comparable to a traditional software library?

2. Is SemVer applicable in the context of Ansible roles? Do you encounter or envision any major challenges in applying the SemVer specifications in this context?

3. Do you follow the SemVer specifications when releasing a new version of your roles?

4. If so, which code changes lead you to create new major, minor, or patch versions?

5. SemVer specifies that the major version number must be incremented when a backward-incompatible change to the public API is made. What would you describe as the public API of a role, and what would constitute a backward-incompatible change to this API?

We contacted a first batch of 22 of the most popular developers in July 2020, with questions 3–5, and received 6 replies. In February 2021, we contacted 15 additional developers with all of the questions, and received 3 replies. We also re-contacted the 6 developers who had replied previously to collect their answers on the first two questions[7], to which we received two replies. Finally, in May 2021, we contacted another 50 developers and sent them a link to an online form containing the same questions. In addition, we contacted 50 organisations which matched the selection criteria listed above, and invited them to share the online form with their Ansible role developers. We also encouraged survey participants to share the form with other role developers to obtain more answers, and shared the form via Twitter. From this last round of invitations, we obtained another 9 replies. Ultimately, we received 14 replies to questions 1–2, and 18 replies to questions 3–5.

Table 5 summarises the developer responses. The remainder of this section describes the responses in more detail.

6.2.1. Applicability of SemVer on Ansible Roles

The vast majority of respondents answer positively on questions 1 and 2. Most developers consider roles to be comparable to general-purpose software libraries, although to varying degrees. One developer highlights that roles can be executed in a playbook in multiple ways, and can be tested using infrastructure test frameworks such as *molecule*, thereby making them very similar to general-purpose libraries. They also generally consider SemVer applicable to Ansible roles, although some developers prefer other versioning formats instead. One developer opines the complete opposite, stating that Ansible roles are "not even a little bit" comparable to general-purpose libraries, on the ground that they are not usable outside of playbooks, and lack complex dependency trees. They further go on to state that "semantic versioning is a ridiculous cult", "barely makes sense for software", and that "literally no one uses it right". This opinion was not shared by any other respondent.

The majority of the respondents encounter no major challenges in applying SemVer to Ansible roles. Nonetheless, we remind the reader that many of the developers to whom we reached out, have a substantial amount of experience in role versioning. Four developers claim to face doubt when selecting the type of version increment to select, mainly in distinguishing between patch and minor releases. One developer mentions that they often have to update their roles when a new version of Ansible is released, and that they are unsure whether that would warrant a major version increment. A final developer states that the API of a role is not well-defined, making it challenging to apply SemVer.

[7]We had not asked these questions in the first round.

Table 5: Summary of top developer responses and their frequency. Due to space constraints, only the 4 most frequent responses are shown.

Question	**Responses (Frequency)**
Roles comparable to libraries?	Yes (*13*) Not at all (*1*)
Applicability of SemVer	Applicable (*12*) Prefer calendar versioning (*1*) Prefer `major.minor` (*1*)
Challenges in applying SemVer	None (*10*) Difficulty choosing bump (*4*) API not well-defined (*1*)
Compliance with SemVer	Yes (*11*) Approximately (*6*) Only the format (*1*)
Reasons for patch bumps	Bug fix (*15*) Optimisations and tweaks (*3*) Fixing typos (*2*)
Reasons for minor bumps	New features (*11*) New variables (*2*) Improvements, optimisations, refactoring (*2*) Other (*2*)
Reasons for major bumps	Removing, renaming, or changing type of variables (*10*) Removing platform or software version support (*4*) Substantial change in behaviour (*3*) Other (*4*)
Role interface	Variables (*10*) Behaviour and responsibilities (*2*) Supported platforms and Ansible versions (*1*)

6.2.2. Adherence to SemVer

All of the 18 respondents confirmed that they are using the SemVer $x.y.z$ format, although compliance with the semantics varies. Six developers mention that they try to follow the specifications, but are not strict about it or try to follow it on a subjective, best-effort basis. They may instead randomly choose a version increment, or may knowingly introduce smaller breaking changes in non-major releases. One developer aligns their role versions with the version of the software it installs, and admits that this approach may not be ideal. Finally, one respondent noted that they solely use the format, but they do not follow its specifications when deciding on a version increment.

6.2.3. Changes Inducing a Version Bump Type

We observe that, when asking developers about the type of version increment that they deem appropriate for a type of changes, most developers agree on what constitutes a patch increment, while consensus on minor increments lessens, and reasons for major increments vary substantially.

The majority of developers agree that patch releases should be limited to simple bug fixes. Some developers also explicitly state that fixing typos is grounds

for a patch release. Moreover, internal optimisations, refactoring, and tweaks to the code are another reason for patch releases, although one developer instead chooses to create a minor release in case the tweak may affect behaviour in edge cases.

Most developers also consider the addition of features and functionality to belong in minor releases, with two developers explicitly stating that this comes in the form of new variables. Other rarely stated reasons for minor releases include internal refactoring and optimisation, improvements to tests, and changes to the configuration.

The respondents generally reserve backward-incompatible changes for major releases, although their definition of backward-incompatibility differ. The most cited breaking change affects default variables, with one developer explicitly qualifying this to only apply to the documented variables. This includes changing the expected type of a variable (e.g., from a string to a list), which would lead to a runtime exception when the value is used. Moreover, renaming or outright removing a variable would lead to a backward-incompatible change as well. One developer mentions that such a removal will not necessarily lead to a runtime exception, since the value is simply ignored. However, they still argue that this leads to a change in behaviour of downstream playbooks, as it may have relied on overriding this variable to customise the role's functionality.

Removing support for a platform, Ansible version, or version of required or installed software is another common reason for breaking changes. Similarly, one developer states that a major version bump of software installed by the role should lead to a major version increment of the role itself. Three developers mention substantial changes in behaviour, or removing features, as reasons for major version bumps. Two developers claim they may also increment the major version when adding support for a new platform or adding a new dependency, respectively. Finally, one developer considers creating a major release to promote the addition of important new functionality, even though no breaking change was made.

Many developers state that it is rare for them to create major version bumps. One developer goes as far as avoiding any backward-incompatible changes, and instead opts to feature-freeze the role and release a completely new role when backward compatibility cannot be ensured. As a result, this developer has not created many major version bumps, but would do so if the role's behaviour changed substantially.

6.2.4. Interface of a Role

The majority of role developers explicitly mention a role's default variables as constituting towards the role's interface. Two developers instead define the interface of a role as its behaviour or responsibilities, i.e., the set of problems it is intended to solve. Although four developers consider removing support for a platform or Ansible version to be a breaking change, only one developer considers this to be an intrinsic part of the role's interface. Finally, one developer mentions that they are using a role which offers its clients the ability to hook into its behaviour at specific points of its control flow. They consider this part of the

role's API, as changes to these hooks can cause changes in behaviour. They also predict that the complexity of Ansible roles and playbooks will greatly increase in the future, leading to more complex and harder to define interfaces.

Findings: All surveyed developers use the SemVer version format, but many do not follow strict specifications. Developers mainly define the API of a role as its default variables, and consider removing or renaming variables to be breaking changes. Changes in the versions of external software, such as supported platforms, installed software, or Ansible itself, may also lead to breaking changes. The majority of developers agree that patch versions are reserved for bug fixes, and minor versions for new features.

6.3. Feature Selection and Classification Model

Our developer survey reveals some indications of developers following rules when incrementing role versions. However, as we only received a reply from 18 developers, and our dataset contains over 2 000, it is impossible to generalise their answers. Moreover, as RQ_2 shows, releases often change multiple directories of a role simultaneously, and there does not appear to be a change type that stands out as a potential indicator of a breaking change or the added functionality. Therefore, in the final part of our empirical analysis, we train a classification model to predict the type of version bump, given the structural difference metrics described in Section 5. Using this model, we can subsequently compare the most important distinguishing features selected by the model to the responses to our developer survey. Moreover, by investigating its precision and recall, we can identify versioning situations where inconsistency or confusion exist.

The dataset thus consists of the 41 structural difference features. In the first part of this analysis, we will train a classification model using *old records*, i.e., version bumps from the previous version of the dataset (cfr. Section 4.6). Our approach to selecting, training, and evaluating our classifier follows the one proposed by Yan et al. [36], who identify features relevant for predicting commits that will be reverted. Subsequently, in RQ_4, we perform a longitudinal study to evaluate whether the trained model is able to predict *new records*, i.e., version bumps that were not yet present in the old dataset.

6.3.1. Model Selection

Our main goal is identifying features and combinations thereof that may indicate which type of bump should be chosen for a new role release. In particular, we focus on the Random Forest classifier, proposed by Ho [37], an ensemble learning classifier which trains multiple Decision Trees simultaneously, and uses either majority voting or average voting to obtain the final classification. This renders it less sensitive to noise [38], which is ideal for our purposes since not all developers consistently follow rules, rendering the dataset to not be the ground truth. Moreover, Random Forest provides the importance of each feature by design, which we can use to derive the desired indicators.

6.3.2. Feature Engineering

Feature engineering [39] is an area of data science that analyses the relation between features and their relevance with respect to the to-be-predicted classes. We use this to discover which features are relevant to our classification problem, and discard all those that are not. Removing such irrelevant features improves the training times and complexity of the model.

We first perform a correlation analysis [40], where we discard correlated features to avoid collinearity. The Spearman correlation coefficient was used to compute the relations among features. Applying a threshold of 0.8 [36], we find two highly-correlated features. The addition of handler blocks and the addition of handler tasks have a high Spearman coefficient of 0.82. We exclude the former from future analysis since the addition of a handler block often implies the further creation of a handler task.

Once the Random Forest model is trained, we can extract the relevance of the features. The Gini impurity index is internally computed for each of the decisions of the trees, enabling one to compute the importance of each feature. Such an index might be influenced by features presenting a high cardinality (i.e., a high number of possible values). Therefore, we perform a second analysis based on permutation importance, where the values of each feature are shuffled while keeping its global distribution. The trained model is applied to this permuted dataset and the final feature relevance is the error difference between the original and permuted datasets, reported in Table 6. When the feature is highly relevant to the classification problem, permuting its values will lead to a large difference in model error, and thus a high permutation error. In contrast, irrelevant features will not cause large changes in the model error after being shuffled, leading to a low error difference.

Then, we perform an ablation study, where we compute the impact of features on the macro F1 score of the model and use this to distinguish relevant features from irrelevant ones. In our previous study, we instead used accuracy as the distinguishing metric since it relies on the correct hits of the classifier. However, the F1 score is more suitable, since it captures both precision and recall. Furthermore, we use a macro metric since this is more resilient to multiclass imbalanced datasets. We use Recursive Feature Elimination (RFE) [41] which recursively removes x features with the lowest relevance until only n features remain. To select the hyper-parameter n, we use an extended approach, namely Recursive Feature Elimination with Cross-Validation (RFECV). Specifically, we use stratified 10-fold cross-validation [42], since this works particularly well on imbalanced datasets. RFECV iteratively tests, using cross-validation, which number of features maximises a specific metric (in our case, macro F1), and applies RFE with the best number of features as n. Thus, this ultimately provides us with the most relevant features, which we summarise in Table 6. Compared to our previous approach, the change in the discrimination hyper-parameter influenced the final selection by incorporating more features. Of the 41 features in the dataset, 1 was excluded due to high correlation and 12 were discarded by the RFECV procedure, leading to a final remaining 28 features. Interestingly, most

Table 6: Overview of structural change categories and permutation-error weight of their corresponding features. Highlighted in bold are those features *excluded* by RFECV and *not* considered in further analysis.

	Addition	Removal	Edit	Reloc.
Dependencies	2.72E-3	1.39E-3	-	-
Platforms	2.46E-2	9.15E-3	-	-
Misc. metadata	-	-	3.19E-2	-
Default var.	7.30E-2	2.23E-2	2.60E-2	**0.0**
Role var.	2.13E-2	3.35E-3	1.15E-2	**4.23E-5**
Default var. file	1.36E-3	**1.84E-4**	-	**7.56E-5**
Role var. file	3.44E-3	1.00E-3	-	**3.24E-4**
Task	7.74E-2	2.06E-2	6.31E-2	4.04E-2
Handler task	1.62E-3	**2.12E-4**	6.29E-3	3.39E-4
Block	8.18E-3	5.16E-3	9.10E-4	7.14E-4
Handler block	-[†]	4.34E-4	**0.0**	**0.0**
Tasks file	8.15E-3	1.09E-3	-	**3.84E-4**
Handler file	**8.82E-4**	**2.71E-4**	-	**0.0**

[†] Excluded by correlation analysis.

of the excluded features mostly concern structural organisation, which may have little to no semantic impact.

Five features stand out in relevance. Adding a task is the highest ranked feature with a score of 7.74×10^{-2}, closely followed by the addition of default variables (7.30×10^{-2}), task edits (6.31×10^{-2}), task relocations (4.04×10^{-2}) and metadata edits (3.19×10^{-2}). We find that changes to tasks and default variables are very important to distinguish different version bumps types. Moreover, one can observe that additions and removals of platforms also carry reasonably high relevance. These observations are consistent with the responses from role developers (cfr. Section 6.2).

6.3.3. Training and Evaluation of the Classifier

Values in the dataset need to be normalised in order to improve the model's performance in later phases of the approach. We employ a normalisation method based on the interquartile range of the data, using the following formula:

$$value = \frac{value - 2Q}{3Q - 1Q}$$

where $1Q$, $2Q$ and $3Q$ represent the first, second, and third quartile values, respectively. This method allows conserving the data distribution in the presence of possible outliers[8].

Using the 28 features and the normalised dataset from the previous step, we train a Random Forest classifier to predict the bump type for each of the

[8] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html

Table 7: Confusion Matrix result from the evaluation of the model. Columns are labelled with the predicted bump type, rows are labelled with the actual bump type.

	Patch*	Minor*	Major*
Patch	44 897	2 867	209
Minor	8 463	6 335	207
Major	1 517	574	1 068

version increments in the dataset. We then evaluate this classifier by having it predict the class for each of these increments. Table 7 depicts the confusion matrix obtained during this evaluation, where rows represent the actual bump type, and columns represent the predicted bump type.

The classifier achieves a precision of 0.82, 0.65, and 0.72 for patch, minor, and major releases, respectively. The accompanying recall scores are 0.94, 0.42, and 0.34 respectively. Globally, our model reports a macro precision and recall of 0.73 and 0.57 respectively, with a macro F1-score of 0.61. In particular, when the release in actuality is a patch, the classifier will likely predict it as such. Moreover, when it predicts a major version bump, it is likely correct. This is offset by the low recall for major, meaning that although a major prediction is likely correct, it is an under-approximation. Furthermore, minor releases show relatively low precision and recall, and the confusion matrix suggests that the classifier often fails to distinguish between minors and patches. This could indicate that the distinction between these two bump types is rather vague, or alternatively, that structural features are insufficient to uncover this distinction. We will investigate potential causes for these erroneous predictions in subsequent research questions.

Findings: Twelve features are irrelevant to distinguish the three types of version bumps. Additions of default variables, task additions, task edits, task relocations and metadata edits are the most significant features. Although our classifier achieves high precision and recall for patch releases, it struggles to distinguish minors from patches. It also significantly under-approximates major releases, although most of its major predictions are correct.

RQ_4: Is our trained model able to correctly predict new instances of version bumps?

The Random Forest classifier which we trained in the previous section is effective at predicting version bumps from structural differences. Recall that this model was trained on *old records*, i.e., version increments in the dataset which were also considered in our previous work [20]. As described in Section 4.6, the extended dataset contains 15 115 new version increments.

We now turn to evaluating the model on these new version increments. Thus, we allow the model to predict the version bump of these new instances. We then compute macro precision, recall, and F1 score, as we did in the previous section, leading to a precision of 0.68, recall of 0.52, and F1 score of 0.56. More

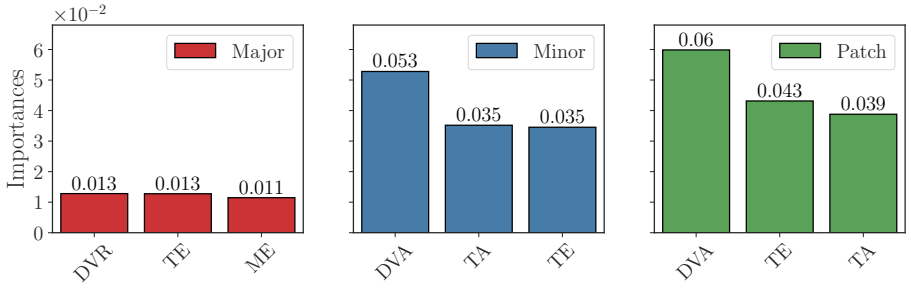


Figure 10: Detailed contribution of the top three features influencing the prediction of a version bump type *(DVA = DefaultVariableAddition, DVR= DefaultVariableRemoval, TA = TaskAddition, TE = TaskEdit, ME = MetaEdit)*.

specifically, we obtain precisions of 0.80, 0.62, and 0.62 for patch, minor, and major, respectively, and recall scores of 0.94, 0.37, and 0.26.

The performance of the model on these new records is worse than what was observed on the old dataset. We thus investigate potential causes of this decrease. To this end, we apply two eXplainable AI (XAI) analyses on our model, to identify the influence of individual features on the model's predictions.

First, we compute Shapley values of each feature for individual predictions of instances using SHAP (SHapley Additive exPlanations) [43]. For a given instance and feature, the Shapley value represents the difference between the predicted value of the instance, and the average predicted value of that instance with random values assigned to the feature, ceteris paribus. Intuitively, when a feature has little relevance in predicting a version bump, random assignments to the feature will not often lead to different predictions, and the computed Shapley value would therefore be low. On the other hand, if a certain feature is very relevant, random assignments will likely lead to different predictions, rendering the computed Shapley value higher. We can then compute global Shapley values by taking the average of the Shapley values for a given feature and all predicted instances of a class.

Figure 10 depicts such Shapley values, computed for the predictions of the new version increments of each version bump type. The sum of all global Shapley values for a feature estimates that feature's contribution to the classification problem. These Shapley values are consistent with the results of RFECV (cfr. Section 6.3.2). However, we can now investigate each feature's contribution to a specific bump type prediction. We can thus observe that changes to default variables are the most relevant features to distinguish between version bump types. For instance, removing such a variable is the most important feature to predict a major release. Moreover, adding variables is the highest ranked feature for both patch and minor increments, suggesting that the type of release depends on the number of such additions. This is in line with the responses from developers, who suggest that a role's interface can be found in its default variables, that their removal is a breaking change, and that new features may

Table 8: Confusion matrix result from the evaluation of the model.

	Patch*	Minor*	Major*
Patch	10137	597	64
Minor	2189	1307	58
Major	346	221	196

come in the form of new variables. Furthermore, recall that the developers mentioned that incrementing the minimum supported Ansible version is a breaking change. Such changes are represented as metadata edits, which is also ranked as an important feature for predicting major increments. Finally, we observe that the most relevant features to predict patch and minor versions are the same. This provides even further evidence that the distinction between these bump types is rather vague. Nonetheless, edits to tasks are slightly more important for patch bumps than minor bumps. This suggests that bug fixes more often change existing tasks, whereas new features may lead to completely new tasks instead.

Secondly, we look into the reasons why mispredictions are made. We thus need to understand which classes are often misclassified, depicted in the confusion matrix in Table 8. We can again see many misclassifications between patches and minors, as was the case in the previous section. These represent 80.17% of all misclassifications of the new instances. We apply Contrastive Explanations, proposed by van der Waa et al. [44], to identify those features that are most often involved in misclassifications between these two classes. This approach works by training a simplified, white-box Decision Tree model, and identifying the decision points in this model which lead an instance to be classified as the predicted class rather than the closest correct classification.

The result of this analysis is depicted in Figure 11, (a) for minor versions predicted as patches, and vice versa in (b). Here, we find that the number of edits to default variables and edits to handler tasks are highly influential in the misclassification of minors as patches. On the other hand, for patch increments predicted as minors, the influencing features are more varied, although edits to tasks and default variables, relocations of tasks, and additions of platforms are most often involved in a misclassification. This suggests that these misclassifications are either due to noise, or due to changes which cannot be fully captured by structural features alone.

Findings: The performance of our trained Random Forest model is considerably lower on newly collected version bumps. The most important features in the classification of version bumps coincide with the responses of developers. The distinction between patch and minor bumps often depends on the number of variables added, tasks added, and tasks edited. Nonetheless, this distinction is vague, since the majority of misclassifications relate to patches predicted to be minor bumps, and vice versa. Many of these misclassifications are heavily impacted by features such as task and default variable edits.

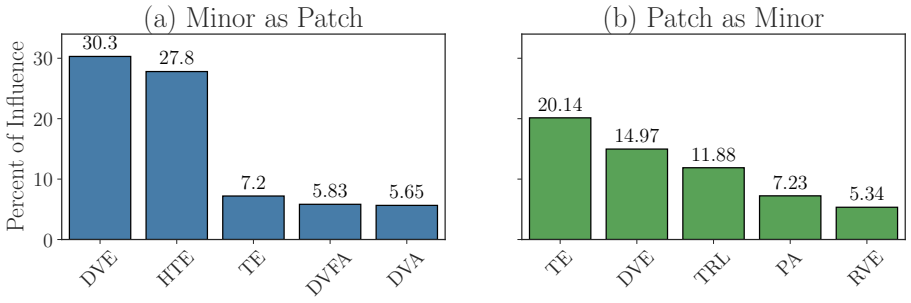


Figure 11: Influence of features on the majority of the wrong predictions. Only the 5 most influential features are shown *(DVA = DefaultVariableAddition, DVE = DefaultVariableEdit, DVR= DefaultVariableRemoval, TA = TaskAddition, TE = TaskEdit, TRL = TaskRelocation, HTE = HandlerTaskEdit, DVFA = DefaultVarFileAddition, PA = PlatformAddition, ME = MetaEdit).*

7. Discussion

7.1. Recommendations

Our findings in RQ_0 show that most Ansible role developers follow the SemVer format. Moreover, the results of RQ_1 show that the actual bump type for a release is not chosen at random, since more severe bumps require larger amounts of effort, which is in line with observations for general-purpose libraries [45]. Indeed, our developer survey reveals that role developers may follow rules to decide which type of bump to apply, although their interpretation does not always align with that of other developers. Moreover, they do not always follow their own rules, and their rules do not always strictly follow the SemVer specifications. Notably, the distinction between patch and minor releases can be ambiguous, since minors may include bug fixes that would otherwise be in a patch. Furthermore, some developers admit to not always incrementing the major version in case of a breaking change. This is consistent with the results of our classifier, which mainly struggles with distinguishing minors from patches, and appears to not recognise some breaking changes that would belong to a major exclusively.

Based on our findings, we can make a number of recommendations to the Ansible community regarding role versioning. Concretely, we propose a set of versioning guidelines (Section 7.1.1), suggest tactics to working with role versions to role clients (Section 7.1.2), and advise role developers to adopt consistent versioning (Section 7.1.3). We also urge the Ansible community as a whole to craft and document the semantics of versioning of Ansible content (Section 7.1.4). Finally, based on our experience obtained from this empirical study, we make a number of recommendations to tool builders (Section 7.1.5).

7.1.1. Versioning Guidelines for Ansible Roles

From the qualitative study, we can extract some general rules to distinguish the SemVer bump types for Ansible roles. Abstractly, one can consider an API

as the set of potential capabilities of its clients. One can then define a backward-incompatible change as any reduction in these capabilities, as there may exist clients who rely on these capabilities. Similarly, expanding these capabilities could be interpreted as the addition of functionality, leading to a minor version bump. Changes that neither expand nor reduce a client's capabilities would then be limited to patches. Note that these definitions may be generally applicable to all types of libraries, not merely Ansible roles.

Concretely, for Ansible roles, we can extract multiple elements that influence a client's capabilities. Most importantly, a role's default variables allow them to influence the behaviour of the role. Moreover, the platforms and Ansible versions supported by the role enable a client to execute the role on various configurations of a machine. If the role installs software onto a machine, the interface of this software is also of importance, as this influences what a client can do with the installed software. Furthermore, since clients can notify a role's handlers from within their own code, these offer additional capabilities. Finally, tags attached to a role's tasks can be selectively enabled or disabled by clients through command-line flags, therefore also impacting the client's capabilities.

Following this definition, we can extract a number of cases where a reduction in client capability, and therefore a breaking change, would occur. These are summarised below.

- **Removing or renaming a default variable** will cause a client's customisation of this variable to be lost.
- **Narrowing the domain of a variable's values**, e.g., by changing its type, may lead to a crash in the client's playbook if it overrides this value.
- **Removing support for a platform** may preclude a client from being able to use the role in its environment. Similar effects may occur when removing support for an **Ansible version** or a **version of required or installed software**.
- **Breaking changes in software** installed by the role could lead to downstream errors in a client's infrastructure. Thus, incrementing the default installed version to a new major version must be considered a breaking change.
- **Reducing the responsibilities** of the role will lead to those responsibilities shifting to the client instead. For example, if a previous version of a role guarantees that a certain package will be installed, but a newer version does not, that would be a breaking change since the clients need to take on this responsibility themselves.
- **Removing handlers**, which can be notified by a client, could lead to a crash in the client's playbook. Similarly, **tags** can be used by clients to cherry-pick tasks to run, and removing such tags leads to a reduction in customisability. Finally, as **role variables** are within the scope of the

client's playbook, removing them may cause issues for clients who use them[9].

The inverse of these cases would create new opportunities for a client to customise the role's behaviour. Therefore, the following changes would lead to an expansion in client capability, new functionality, and minor version bumps.

- Adding default variables or widening their domain. However, note that adding default variables which are *required* to be overridden by a client, may be a breaking change.
- Adding support for a platform, Ansible version, or required or installed software.
- New functionality (i.e., minor version increments) in installed software.
- Expanding the responsibilities of the role.
- Adding *documented* handlers, tags, and role variables to be used by the client.

Finally, changes that do not affect a role's clients should be considered patch version increments. The following are some examples.

- Code reorganisation, refactoring, and optimisations.
- Changes to documentation.
- Changes to tests.
- Bug fixes which do not break backward compatibility.

However, these are likely over-simplifications, as a version bump is rarely limited to changes in one directory (RQ_2 and RQ_3) and not all developers necessarily follow such rules consistently. Nonetheless, our classifier's selection of features (cfr. Section 6.3.2 and RQ_4) suggests a similar idea, where the addition and removal of tasks, blocks, and variables are selected as important features to distinguish version types, as well as the addition and removal of platforms. These features also coincide with the most common changes to the role (cfr. Table 4).

7.1.2. Recommendations for Role Clients

For clients who include a role in their playbooks, as well as for roles that depend on other roles, we first and foremost recommend to **pin the dependency's version**. Otherwise, Ansible will default to installing the latest version of the dependency. This may lead to **idempotency issues** when different versions of the dependency are used between playbook runs.

[9]This guideline may be narrowed to exclude handlers, tags, and role variables that are undocumented.

Second, when deciding whether to upgrade a role dependency, we recommend to **avoid making assumptions** about the changes that went into the new version **based on the version number alone**. Instead, one should at least **read the changelog** to determine whether the upgrade may contain breaking changes or new functionality, since our developer survey shows that some role developers do not strictly adhere to SemVer.

Finally, as some breaking changes such as removing a variable do not necessarily lead to crashes in a client's playbook, we recommend to **thoroughly test a playbook's behaviour**. Ideally, playbooks should regularly be tested, even if no changes have been made, so that any issues caused by upstream dependencies can be caught early. It is also recommended to **re-execute playbooks frequently** to prevent configuration drift [46], and this can also help to detect potential issues with dependencies as soon as possible.

7.1.3. Recommendations for Role Developers

Of the 8 289 roles in active development (cfr. Section 4.2), we find that 3 223 (38.5%) have not released any version. Moreover, merely 4 849 (58.5%) have released a version that syntactically adheres to the SemVer format throughout their lifetime. Therefore, we urge role developers to **adopt versioning practices, specifically Semantic Versioning**, in their roles, if they have not already. Although SemVer can be a burden on developers, it provides substantial benefits to role clients, such as conveying the nature of changes to the role, and providing the ability to select and pin specific versions to install. Furthermore, we recommend developers to take inspiration from the proposed versioning guidelines (Section 7.1.1) to select the appropriate version bump. We also advise developers to avoid creating patch releases consisting solely of changes that do not affect the role's behaviour, such as changes to tests, Continuous Integration (CI), or simple refactoring, to prevent clients from having to fruitlessly upgrade their dependencies.

If unable to migrate to SemVer, e.g., because it would be too problematic to switch from a legacy versioning format, we recommend role developers to **clearly document the versioning strategy adopted**. Such documentation should at least include a description of what each type of version bump may entail. Similarly, for role developers that adopt a custom, SemVer-compliant versioning scheme, such as including the version of installed software in the build metadata, such "extensions" should be clearly described, and ideally, applied consistently.

We also recommend to **thoroughly test roles with example playbooks**, ideally in a Continuous Integration (CI) environment. This will enable the developer to better understand the effect of certain changes on clients of the role. Moreover, it may potentially uncover backwards-incompatible changes that are not immediately apparent. If a change to the role causes a test playbook to fail, that change could also cause client playbooks to fail, and would therefore be backwards-incompatible.

Finally, it is worth highlighting a key difference between traditional software libraries and Ansible roles. In traditional libraries, addition of functionality

often comes in the form of new classes, methods, or functions for a client to use. Clients need to specifically opt in to using these new features, by adapting their code to use the new classes, methods, or functions. On the contrary, for roles, new functionality may come in the form of new tasks that cover new responsibilities. If these new tasks execute by default, the new behaviour may cause conflicts with responsibilities already covered by the client. Therefore, we advise role developers to **disable new responsibilities by default**, and to require clients to explicitly opt in to the new behaviour using a feature flag.

7.1.4. Recommendations for the Ansible Community

Our results show that Ansible role developers do not always adhere to the Semantic Versioning guidelines (cfr. Section 6.2). They may instead release backwards-incompatible changes as patch or minor releases, and deciding which version bump to choose can be a difficult choice to some. This may in part be due to a lack of guidelines when it comes to applying versioning on roles.

We therefore urge the Ansible community to construct a **set of specific guidelines** on how to apply SemVer to Ansible roles. We recommend the community to take inspiration from the guidelines proposed above, which have been crafted using responses from popular Ansible role developers. Moreover, we advise to showcase real-world examples of versioning of Ansible roles, accompanied by the reasoning behind choosing the given version bump. This would aid in steering role developers towards versioning semantics which are applied consistently throughout the ecosystem. We believe such consistency would be beneficial to the experience of role clients.

It is important to note that, during the time in which this research was conducted, it has become increasingly clear that Galaxy roles will soon be superseded by collections [47, 48], with Ansible maintainers advising against creating new stand-alone roles, and recommending role developers to migrate to collections instead [49]. Although this research focuses solely on stand-alone roles, we believe that our results and recommendations are equally applicable to collections. In fact, we believe our findings are of the upmost importance for collections, since semantic versioning of collections is mandatory rather than merely recommended. Consequently, without proper guidelines in place, the number of versions that disregard the semantics of SemVer will undoubtedly rise. We believe the upcoming migration to collections is an opportune time to rectify the shortcomings of versioning of the past, and therefore urge the community as a whole to **formally define the semantics of semantic versioning in Ansible content**.

7.1.5. Recommendations for Researchers and Tool Builders

Our results suggest that many changes to roles merely consist of textual refactorings (cfr. RQ_3), which may be of no interest to the study at hand. Therefore, we advise researchers to **look beyond the lexical or syntactical level**, and instead leverage structural or semantic representations for Ansible role analysis.

Furthermore, we believe the guidelines proposed above capture the broad basics of versioning in Ansible roles. We recommend tool builders to **employ these guidelines in new tooling** for Infrastructure-as-Code practitioners. For example, a tool that recommends an Ansible role developer the appropriate version bump based on the changes applied to the role, could aid the developer in understanding and adhering to SemVer. These guidelines could similarly be used in a tool which checks for version compatibility, which would help role clients to manage their dependencies.

7.2. Validity of the Structural Model

Because a large portion of our empirical study makes use of structural changes distilled from role version increments, it is of the upmost importance that these distilled changes sufficiently capture actual changes. Naturally, since these changes are distilled from our structural model, we must also ensure that our structural model correctly represents the role. Therefore, we manually assessed the validity of the structural model and distilled changes using 100 randomly-sampled version increments.

To validate the structural model, we manually investigated any differences between the original YAML documents of a role and the extracted structural model, for both the old and new versions in each of the 100 version increments. To facilitate this process, we converted our structural model representation back to a YAML document that would be understood by Ansible, after which we could apply standard textual differencing techniques to uncover any discrepancies. Throughout all of the role versions, we observed no unexpected differences between these two representations. Instead, the observed differences were limited to deliberate syntactical generalisations.

Afterwards, for each version increment in the sample, we compared the distilled structural changes to the textual changes produced by the `git diff` command. We found that our structural change distiller sufficiently captures all changes to files considered in the structural model. It occasionally failed to recognise that a task or block was edited, and instead extracted a pair of removal and addition changes for the edited element. However, the two versions of the structural element were substantially different, and therefore did not exceed the similarity threshold to be considered the same element. Moreover, the inability to distinguish between edits and pairs of additions and removals is an inherent limitation of after-the-fact change distilling, and only occurred in very few cases. Therefore, we are confident in the accuracy of the distilled structural changes.

Finally, as described in Section 5.1, our structural model only represents YAML files in the five role directories that contain the role's code. However, roles may contain additional directories, such as *files*, *templates*, or directories containing tests. The content of these directories is not considered by the structural model. Consequently, any change to this content is not extracted by the change distiller. Although this is a limitation of the approach, during the inspection of the 100 version increments, we only found two cases where edits to *files* or *templates* directories were not accompanied by a related change to the

Table 9: Macro precision and recall scores of Random Forest models after applying various undersampling techniques to balance the dataset.

	Patch		**Minor**		**Major**	
	P	R	P	R	P	R
Baseline	0.82	0.94	0.65	0.42	0.72	0.34
Random	0.85	0.73	0.46	0.47	0.19	0.58
NM-v1	0.88	0.46	0.20	0.17	0.09	0.76
NCR	0.81	0.86	0.49	0.42	0.48	0.40
Tomek Links	0.82	0.94	0.65	0.42	0.70	0.34

role's tasks or variables. Furthermore, the role's tests are irrelevant to the role's behaviour, and therefore changes to tests are not important for our purposes. Hence, we do not believe this limitation has a large negative influence on our findings.

7.3. Effects of Data Balancing on the Classifier

Our dataset is highly imbalanced, showing a clear bias towards patch increments (cfr. RQ_1). This is to be expected when analysing semantic version bumps. Consequently, the trained model may exhibit bias towards the majority class (patch) when predicting the version bump type. Therefore, we investigated whether data balancing can eliminate some of this bias and improve the trained model.

There exist two general approaches to deal with imbalanced datasets, namely *oversampling* and *undersampling*. The former approach creates synthetic instances for under-represented classes, whereas the latter approach removes instances from the majority class based on certain filtering criteria. As we prefer to predict version bumps based on non-synthetic and unique instances, we only consider the latter approach. We selected four undersampling techniques, namely random undersampling, which randomly removes instances from the majority class until perfect balancing is achieved; NearMiss, proposed by Mani and Zhang [50] using the first distance metric (NM-v1); the Neighbourhood Cleaning Rule (NCR) proposed by Laurikkala [51]; and Tomek Links [52]. We refer the reader to the cited material for a description of these approaches.

We apply each of these approaches separately to train and evaluate a Random Forest model. To this end, we apply stratified 10-fold cross-validation in a similar manner to the approach presented in Section 6.3.3, but apply balancing on the 9 training folds, while the testing fold is left unmodified. We then compare the precision and recall obtained for each bump type against the baseline model trained without data balancing. The obtained results are depicted in Table 9.

We observe that all balancing approaches achieve worse results than the baseline, with the Tomek Links approach achieving the closest results, yet still being marginally outperformed by the model trained on the unbalanced dataset. Although some approaches outperform the baseline on specific aspects, their performance on other aspects is lacklustre. For example, although the NearMiss

Table 10: Summary of the manual investigation of the model's misclassifications. Columns indicate the predicted version bump type, while rows depict the actual bump type. Cells denote the number of model predictions which were found to be correct, but where the actual bump type disagrees with the proposed guidelines.

	Patch*	Minor*	Major*
Patch	-	8	12
Minor	8	-	20
Major	7	3	-

approach attains substantially higher recall on major version increments, it is offset by a very poor precision score on the same class. Therefore, we conclude that applying data balancing on our dataset does not improve our results.

7.4. Model Misclassifications

As developers may not all follow the same rules, our dataset may contain incorrectly tagged versions. Hence, our classifier may pick up on incorrect rules. Furthermore, structural changes may not always be sufficient to distinguish release types. Therefore, we manually investigate a sample of version bumps that were incorrectly classified in the longitudinal study (RQ_4).

We randomly sampled 20 version bumps for each type of misprediction, leading to a total of 120 version bumps to inspect. We then manually classify each version bump according to the guidelines proposed in Section 7.1.1. Afterwards, we compare our manual classification to the version bump type predicted by the model, as well as to the version bump type originally chosen by the developer. This enables us to assess whether the model is able to correctly capture the various proposed guidelines, and to identify potential reasons for misclassification. The result of this manual investigation is summarised in Table 10.

We again observe that developers often struggle with choosing between patch and minor version bumps. For minor versions that were misclassified as patches, we find that 8 contained no new features or functionality, and were instead limited to test changes, refactorings, or simple bug fixes. We also found 4 instances which actually contained breaking changes, such as removing variables. Similarly, 8 of the patches that were predicted to be minor versions contained new variables or added support for new platforms. Moreover, 6 other version increments removed support for platforms, or incremented the minimum required Ansible version, although the classifier could not identify these breaking changes.

For both patch and minor releases which the model classified as major releases, we find that a large number of them in fact contain breaking changes, which the model correctly identified. All of the sampled minor version bumps remove variables or support for platforms, leading to all 20 of them to be correctly classified by the model, although incorrectly bumped by the role developer. Similarly, 12 of the patch versions remove variables or increment the minimum required Ansible version, which are changes that should lead to a major version increment according to the proposed guidelines.

Interestingly, although the model can effectively identify breaking changes in patch and minor releases, it often failed to identify these same changes in the major releases. Of the 20 sampled major bumps that were misclassified as patches, only 7 contained refactoring or were limited to simple bug fixes. One of these was in fact using date-based versioning, and the turn of the calendar year made it appear to be a major version bump. Similarly, only 3 of the 20 sampled major bumps that were predicted to be minor versions contained no breaking changes.

We found that the vast majority of the model's misclassifications relate to distinguishing between patch and minor version bumps (cfr. RQ_4). Our manual investigation suggests that this can be partially attributed to noise in the dataset, caused by uncertainty of the developer. However, the misclassifications may also be caused by the absence of semantic information in the structural features. For example, a developer may add new tasks to a role to fix a bug, but such additions can also happen when new functionality is introduced. In particular, in RQ_4, we found that the main causes of the model's confusion between the two is due to additions of tasks, as well as task edits. Moreover, since minor releases can also contain bug fixes, the relevant features to predict these increment types often overlap. Nonetheless, even with semantics in place, distinguishing between these cases may prove difficult, since such decisions may be subjective. For example, one developer may consider adding a task to install a new configuration file as the addition of a feature, whereas another developer may see this as a bug fix if the absence of this file caused issues for some clients. Taken to the extreme, one may even consider certain bug fixes to be breaking changes when a client depended specifically on the erroneous implementation.

8. Threats to Validity

The empirical nature of our research exposes its findings to potential threats to validity. We present them following the classification and recommendations of Wohlin et al. [53].

A threat to *construct validity* comes from the way we designed the structural model. It does not consider the *files* and *templates* directories, which could contain significant changes. However, our developer survey suggests that the most important changes relate to its interface, i.e., default variables. Moreover, we manually validated the structural model and structural change distiller (cfr. Section 7.2), and found that changes to these directories very rarely occur in isolation. Instead, they are almost always accompanied by a related change to the role's tasks or variables, all of which are captured by the change distiller. Therefore, we believe this limitation has no significant impact on our findings.

As a threat to *internal validity*, we did not consider all version bumps of Ansible roles, since we removed tagged version numbers such as pre-releases. This may have partially influenced our results. However, we are interested in stable versions only, and thus, this filtering does not affect our findings. Moreover, the gathered dataset is not the ground truth, and may contain noise. To alleviate this issue, we used feature elimination and majority voting to be more

resistant to noise, and inspected our classifier's mispredictions (Section 7.4). Finally, our dataset is highly imbalanced, showing a clear bias towards patch releases which are significantly more common. We address this issue by using stratified cross-validation, which maintains the distribution of the original data in the constructed folds. We also investigated the use of data balancing, but found that it did not improve the results (Section 7.3).

As a threat to *external validity*, we cannot claim that our findings generalise to Infrastructure-as-Code projects for platforms other than Ansible. Furthermore, we cannot be certain that the findings of our developer survey generalise to all Ansible role developers due to the relatively low number of respondents. Nevertheless, the survey respondents have varying levels of popularity and experience with role versioning.

9. Conclusion

Like general-purpose libraries, Ansible roles need to be versioned to provide new role releases to its clients. Although Ansible recommends semantic versioning, it is unclear what the meaning of patch, minor, and major releases are in roles. In this paper, we empirically investigated the state of versioning in Ansible roles. From a dataset of over 8 500 roles and over 90 000 versions, we found that most developers use the SemVer scheme and that development practices are consistent with observations in general-purpose libraries. We designed a structural model for Ansible roles, and created a domain-specific change extraction algorithm to extract structural changes between two version of a role. We found that many textual changes between releases do not incur a structural change, and that many role releases change multiple role elements. We then trained a Random Forest classifier to predict the type of version bump, given 41 features in the form of metrics of the structural difference between the two versions. Its selection of features highlights key indicators to distinguish different version bumps, with the addition of default variables and tasks standing out. Furthermore, the classifier's results suggest that the distinction between patch and minor version bumps is often unclear, and that breaking changes do not always strictly lead to a major version. We confirmed these findings with a qualitative developer survey, where we question 18 popular role developers regarding SemVer-compliance and the changes that trigger them to do a certain release. Finally, we extracted two general guidelines, namely that the addition of capabilities for the client should lead to a new minor version, and conversely, that the removal of such capabilities is a breaking change that should lead to a major increment. In particular, removing a role's default variables or dropping support for a certain operating system version are two main examples of potential backward-incompatible changes. Conversely, adding such elements can be considered new functionality, leading to the possibility of a minor version increment. Although many developers appear to follow such guidelines implicitly, they do not do so consistently.

Acknowledgements

This research was partially funded Research Foundation – Flanders (FWO) (grant number 1SD4321N) and by the Excellence of Science project 30446992 SECO-Assist financed by FWO-Vlaanderen and F.R.S.-FNRS.

References

[1] Ansible collections overview, https://github.com/ansible-collections/overview/blob/f3b771be9c8727e730922107f74493c519665ed4/README.rst#where-we-ve-come-from-and-where-we-are-going, accessed: 2021-06-03 (2020).

[2] F. Fontein, Inclusion of cyberark.conjur in Ansible 2.10, https://github.com/cyberark/ansible-conjur-collection/issues/30, accessed: 2021-06-03 (2020).

[3] F. Fontein, Including dellemc.openmanage collections into Ansible 3.0.0, https://github.com/ansible-collections/ansible-inclusion/discussions/3#discussioncomment-272508, accessed: 2021-06-03 (2021).

[4] R. Wimmer, githubixx.kubectl readme: Versions, https://github.com/githubixx/ansible-role-kubectl/blob/60f7da8579935295da756d1bcb0a2e4e12a290a1/README.md#versions, accessed: 2021-06-03 (2017).

[5] S. Raemaekers, A. van Deursen, J. Visser, Semantic versioning and impact of breaking changes in the maven repository, Journal of Systems and Software 129 (2017) 140–158. doi:10.1016/j.jss.2016.04.008.

[6] T. Appnel, Comment on "Add 'update' option to ansible-galaxy", https://github.com/ansible/ansible/issues/6466#issuecomment-234387668, accessed: 2021-06-03 (2016).

[7] H. Preston, https://twitter.com/hfpreston/status/1330257384133029891, accessed: 2021-06-03 (2020).

[8] J. Geerling, Comment on "Ansible is stressing me out more than doing anything manually (and I eventually do something manually)", https://www.reddit.com/r/ansible/comments/ist96e/c/g5a8ib1/, accessed: 2021-06-03 (2020).

[9] J. Geerling, Add 'update' option to ansible-galaxy, https://github.com/ansible/ansible/issues/6466, accessed: 2021-06-03 (2014).

[10] B. Coca, role versioning, https://github.com/ansible/proposals/issues/23, accessed: 2021-06-03 (2016).

[11] X. Manning, PyratLabs.k3s changelog, https://github.com/PyratLabs/ansible-role-k3s/blob/f46450319b6ba79db44cba7a2750cccbca5bc65d/CHANGELOG.md#breaking-changes-1, accessed: 2021-06-03 (2020).

[12] elastic.elasticsearch changelog, https://github.com/elastic/ansible-elasticsearch/blob/82c2129f174e744ed6013d3d5dae863192f54a92/CHANGELOG.md#breaking-changes, accessed: 2021-06-03 (2019).

[13] J. Guldmyr, Documenation or fix when using ansible-role-postfix v1.1.0, https://github.com/fgci-org/fgci-ansible/issues/142, accessed: 2021-06-03 (2016).

[14] O. Faměra, OndrejHome.ha-cluster-pacemaker version history, https://github.com/OndrejHome/ansible.ha-cluster-pacemaker/tags, accessed: 2021-06-03 (2021).

[15] M. Guerriero, M. Garriga, D. A. Tamburri, F. Palomba, Adoption, support, and challenges of infrastructure-as-code: Insights from industry, in: Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME19), Industrial Track, 2019, pp. 580–589. doi:10.1109/ICSME.2019.00092.

[16] A. Rahman, C. Parnin, L. Williams, The seven sins: Security smells in infrastructure as code scripts, in: Proceedings of the 41st International Conference on Software Engineering (ICSE19), 2019, pp. 164–175. doi:10.1109/ICSE.2019.00033.

[17] E. V. der Bent, J. Hage, J. Visser, G. Gousios, How good is your Puppet? an empirically defined and validated quality model for Puppet, in: Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER18), 2018, pp. 164–174. doi:10.1109/SANER.2018.8330206.

[18] S. Dalla Palma, D. Di Nucci, F. Palomba, D. A. Tamburri, Toward a catalog of software quality metrics for infrastructure code, Journal of Systems and Software 170 (2020) 110726. doi:10.1016/j.jss.2020.110726.

[19] R. Opdebeeck, A. Zerouali, C. De Roover, Andromeda: A dataset of Ansible Galaxy roles and their evolution, in: Proceedings of the 2021 International Conference on Mining Software Repositories (MSR21), 2021, pp. 580–584.

[20] R. Opdebeeck, A. Zerouali, C. Velázquez-Rodríguez, C. De Roover, Does Infrastructure as Code adhere to Semantic Versioning? An analysis of Ansible role evolution, in: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM20), IEEE, 2020, pp. 238–248. doi:10.1109/SCAM51674.2020.00032.

[21] A. Rahman, R. Mahdavi-Hezaveh, L. Williams, A systematic mapping study of infrastructure as code research, Information and Software Technology 108 (2019) 65–77. doi:10.1016/j.infsof.2018.12.004.

[22] W. Hummer, F. Rosenberg, F. Oliveira, T. Eilam, Testing idempotence for infrastructure as code, in: Proceedings of the 14th International Middleware Conference (Middleware13), 2013, pp. 368–388. doi:10.1007/978-3-642-45065-5_19.

[23] R. Shambaugh, A. Weiss, A. Guha, Rehearsal: A configuration verification tool for Puppet, in: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI16), 2016, pp. 416–430. doi:10.1145/2908080.2908083.

[24] K. Ikeshita, F. Ishikawa, S. Honiden, Test suite reduction in idempotence testing of infrastructure as code, in: Proceedings of the 11th International Conference on Tests and Proofs (TAP17), 2017, pp. 98–115. doi:10.1007/978-3-319-61467-0_6.

[25] A. Rahman, L. Williams, Source code properties of defective infrastructure as code scripts, Information and Software Technology 112 (2019) 148–163. doi:j.infsof.2019.04.013.

[26] T. Sharma, M. Fragkoulis, D. Spinellis, Does your configuration code smell?, in: Proceedings of the 13th International Conference on Mining Software Repositories (MSR16), 2016, pp. 189–200. doi:10.1145/2901739.2901761.

[27] S. Dalla Palma, D. Di Nucci, F. Palomba, D. A. Tamburri, Within-project defect prediction of Infrastructure-as-Code using product and process metrics, IEEE Transactions on Software Engineering (2021) 1–1doi:10.1109/TSE.2021.3051492.

[28] C. Bogart, C. Kästner, J. Herbsleb, F. Thung, How to break an API: Cost negotiation and community values in three software ecosystems, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE16), ACM, 2016, pp. 109–120. doi:10.1145/2950290.2950325.

[29] C. Bogart, A. Filippova, C. Kästner, J. Herbsleb, Survey of ecosystem values, http://breakingapis.org/survey/, accessed: 28/10/2017.

[30] A. Decan, T. Mens, What do package dependencies tell us about semantic versioning?, IEEE Transactions on Software Engineering 47 (6) (2019) 1226–1240. doi:10.1109/TSE.2019.2918315.

[31] J. Dietrich, D. Pearce, J. Stringer, A. Tahir, K. Blincoe, Dependency versioning in the wild, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR19), IEEE, 2019, pp. 349–359. doi:10.1109/MSR.2019.00061.

[32] B. Fluri, M. Wuersch, M. PInzger, H. Gall, Change distilling: Tree differencing for fine-grained source code change extraction, IEEE Transactions on Software Engineering 33 (11) (2007) 725–743. doi:10.1109/TSE.2007.70731.

[33] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Montperrus, Fine-grained and accurate source code differencing, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE14), 2014, pp. 313–324. doi:10.1145/2642937.2642982.

[34] R. Stevens, C. De Roover, Extracting executable transformations from distilled code changes, in: Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER17), 2017, pp. 171–181. doi:10.1109/SANER.2017.7884619.

[35] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, L. Devine, Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices?, in: Annual Meeting of the Southern Association for Institutional Research, 2006.

[36] M. Yan, X. Xia, D. Lo, A. E. Hassan, S. Li, Characterizing and identifying reverted commits, Empirical Software Engineering 24 (4) (2019) 2171–2208. doi:10.1007/s10664-019-09688-8.

[37] T. K. Ho, The random subspace method for constructing decision forests, IEEE Transactions on Pattern Analysis and Machine Intelligence 20 (8) (1998) 832–844. doi:10.1109/34.709601.

[38] Tin Kam Ho, Random decision forests, in: Proceedings of the 3rd International Conference on Document Analysis and Recognition (ICDAR95), 1995, pp. 278–282. doi:10.1109/ICDAR.1995.598994.

[39] A. Zheng, A. Casari, Feature engineering for machine learning: principles and techniques for data scientists, O'Reilly Media, Inc., 2018.

[40] M. Hazewinkel, Correlation in statistics, in: Encyclopaedia of Mathematics, Kluwer, 2001.

[41] I. Guyon, J. Weston, S. Barnhill, V. Vapnik, Gene selection for cancer classification using support vector machines, Machine Learning 46 (2002) 389–422. doi:10.1023/A:1012487302797.

[42] K. Sechidis, G. Tsoumakas, I. Vlahavas, On the stratification of multi-label data, Machine Learning and Knowledge Discovery in Databases (2011) 145–158doi:10.1007/978-3-642-23808-6_10.

[43] S. M. Lundberg, S.-I. Lee, A Unified Approach to Interpreting Model Predictions, in: I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems 30, Curran Associates, Inc., 2017, pp. 4765–4774.

[44] J. van der Waa, M. Robeer, J. van Diggelen, M. Brinkhuis, M. Neerincx, Contrastive Explanations with Local Foil Trees, in: 2018 Workshop on Human Interpretability in Machine Learning (WHI18), 2018.

[45] A. Zerouali, T. Mens, J. Gonzalez-Barahona, A. Decan, E. Constantinou, G. Robles, A formal framework for measuring technical lag in component repositories—and its application to npm, Journal of Software: Evolution and Process 31 (8) (2019) e2157. doi:10.1002/smr.2157.

[46] K. Morris, Infrastructure as Code: Managing Servers in the Cloud, O'Reilly, 2016, Ch. 8, pp. 133–150.

[47] S. Sbarnea, Are standalone Ansible roles a dead-end?, `https://www.reddit.com/r/ansible/comments/lnbclr/are_standalone_ansible_roles_a_deadend/`, accessed: 2021-06-09 (2021).

[48] S. Doran, Comment on "Feature request: make "version" a valid role metadata attribute", `https://github.com/ansible/ansible/issues/13496#issuecomment-834519928`, accessed: 2021-06-09 (2021).

[49] J. R. Barker, Ansible Community Galaxy next steps (help needed), `https://www.reddit.com/r/ansible/comments/na4end/ansible_community_galaxy_next_steps_help_needed/`, accessed: 2021-06-09 (2021).

[50] I. Mani, I. Zhang, kNN Approach to Unbalanced Data Distributions: A Case Study involving Information Extraction, in: Proceedings of the ICML'2003 Workshop on Learning from Imbalanced Datasets, 2003.

[51] J. Laurikkala, Improving identification of difficult small classes by balancing class distribution, in: Conference on Artificial Intelligence in Medicine in Europe, Springer, 2001, pp. 63–66. doi:10.1007/3-540-48229-6_9.

[52] I. Tomek, Two Modifications of CNN, IEEE Transactions on Systems, Man, and Cybernetics SMC-6 (11) (1976) 769–772. doi:10.1109/tsmc.1976.4309452.

[53] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, A. Wesslen, Experimentation in Software Engineering - An Introduction, Kluwer, 2000.