Does Infrastructure as Code Adhere to Semantic Versioning? An Analysis of Ansible Role Evolution

Ruben Opdebeeck
ropdebee@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Ahmed Zerouali
azeroual@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Camilo Velázquez-Rodríguez
cavelazq@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Coen De Roover
cderoove@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

***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. As such, 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. We collect a dataset of over 70 000 version increments spanning upwards of 7 800 Ansible roles. Moreover, 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 components. Then, using the structural difference metrics, we train a Random Forest classifier to predict applicable version bumps for Ansible role releases. Lastly, 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.**

***Index Terms*—Ansible; Infrastructure as Code; Semantic Versioning; empirical study; mining software repositories**

I. 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, an online registry containing upwards of 20 000 roles, contributed by third-party developers, which users 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 software, 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)[1] 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.

However, these rules are designed for general-purpose libraries, and do not readily apply to Ansible roles. The first SemVer specification requires that the software has a public API, whereas the concept of APIs in Ansible roles is not formally defined. Moreover, although Ansible recommends the SemVer *format*, it makes no mention of its rules, opening up possibilities for developer practices to diverge. Consequently, this may lead to confusion for clients and developers alike.

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 version bump type. A better understanding of these changes would be beneficial to the clients of a role to identify potential costs in updating an installed role, as well as for new role developers to align their role versioning with established practices.

There have been multiple studies investigating IaC (e.g., [1]–[4]). 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] https://semver.org/

- We collect a dataset containing upwards of 7 800 roles, 80 000 role versions, and 70 000 version increments.
- 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.
- We carry out quantitative and qualitative studies to shed light on role developer practices regarding SemVer.
- Finally, using the extracted structural changes, we develop a classifier to predict the applicable SemVer version bump type for a new role release.

The dataset we have collected is available at https://doi.org/10.5281/zenodo.4039514. Additionally, our tools for analysis are available in a replication package found at https://doi.org/10.5281/zenodo.4040647.

II. 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.

A. Infrastructure as Code

Infrastructure as Code is an emerging research domain, with an increasing number of works published each year [5]. Industrial IaC practitioners often face the challenge of identifying defects in their files [1]. 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 [6]–[8]. Other researchers have focused on syntactical properties, metrics, smells, and detection rules to highlight potential problems [2]–[4], [9], [10]. For example, Rahman and Williams [9] construct defect prediction models using 10 source code properties. Sharma et al. [10] propose a catalogue of 24 design and implementation smells for IaC code. Van der Bent et al. [3] define and empirically validate a suite of maintainability metrics for Puppet code. Building upon this, Dalla Palma et al. [4] propose 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.

B. Semantic Versioning

The semantic versioning specifications are commonly recommended by package managers to denote the kind of changes in new releases of software packages. Because of its importance, SemVer has been subjected to many research studies. Raemaekers et al. [11] 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.

```
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: An example of an Ansible playbook.

Bogart et al. [12] 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 [13]. They observe 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 observe that the frequency of breaking changes is higher in some ecosystems (*npm*, *Rust*) than others (*Perl*, *CRAN*, *Eclipse*).

Decan et al. [14] empirically studied SemVer compliance in four ecosystems (*Cargo*, *npm*, *Packagist*, and *Rubygems*) by analysing package dependency constraints. They find that the proportion of compliant dependency constraints increases over time in all ecosystems, and identify factors that influence the degree of compliance. Similarly, Dietrich et al. [15] 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.

III. 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. As such, 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.

A. 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 execute 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.

B. 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. Each of the subdirectories must contain a `main.yml` file, which Ansible loads by default, but the subdirectory may be omitted if unused. Subdirectories can contain supplementary files, which can be loaded through special actions by the role's tasks.

The following are standard directories for roles.

- Files in the `tasks` directory contain the role's blocks and tasks, whereas the `handlers` directory contains the role's handlers.
- 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 user, and are therefore often used as constants. Contrarily, default variables are much easier to override, and serve as the means for the user to parametrise the role's behaviour. To avoid confusion, we shall name role variables "constant variables" throughout the remainder of this paper.
- 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. These directories do not require the `main.yml` file.
- Finally, the `meta` directory's main file contains metadata for the role, such as author, description, license, etc. It 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. This directory can contain other files, but they are ignored.

TABLE I
OVERVIEW OF THE COLLECTED DATASET.

Stage	#Roles	#Authors	#Versions	#Incr.
Role discovery	24 620	5 979	N/A	N/A
Role collection	23 681	5 665	N/A	N/A
Version extraction	7 823	1 891	80 997	72 559
Struct. model constr.	7 248	1 779	72 642	64 542

To ease the discovery of third-party roles, Ansible provides Galaxy[2], a central registry currently containing over 20 000 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 its users 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.

IV. Method and Data Extraction

To investigate the versioning of Ansible roles, we gathered a dataset of 7 823 unique Galaxy roles written by 1 891 unique authors, in total containing 72 599 version increments. This section describes the pipeline used to collect this data and extract the necessary information. Figure 1 depicts an overview of this pipeline, while Table I 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 syntactical difference metrics, e.g., the number of lines inserted and deleted, for each version bump. However, syntactical differences are sensitive to semantically-irrelevant implementation details such as refactorings, 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 V. 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, whereas another was removed.

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 perform extensive

[2]https://galaxy.ansible.com

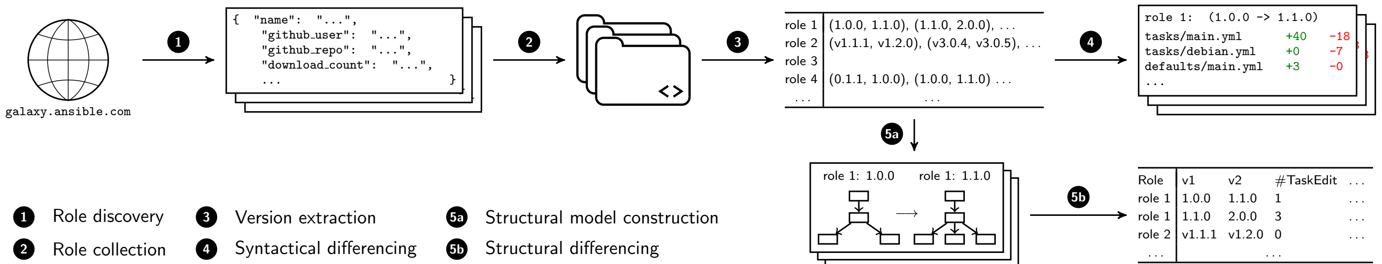

Fig. 1. Overview of the data extraction pipeline.

filtering, although we do removed roughly 300 roles that were marked as deprecated. Ultimately, we extracted 24 620 roles written by 5 979 unique authors, representing a complete listing of non-deprecated roles on Galaxy as of June 12th, 2020. 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.

2) Role Collection: In the second stage, we clone the git repositories of each collected role through the GitHub URL obtained from Galaxy. Of the 24 620 roles gathered in the previous phase, we failed to clone the repository of 939 because they were not available, these roles are thus removed from further processing. This reduces the size of our dataset to 23 681 roles, contributed by 5 665 authors.

3) Version Extraction: We then move on to extracting role versions from the cloned repositories' tags. We extract each tag from every role repository and retain only those that match a lenient derivative of the SemVer format that allows an arbitrary number of version parts (e.g., $x.y$, $x.y.z$, $x.y.z.a$, etc.). We allow for non-SemVer versions to later investigate the prevalence of the SemVer format, while still filtering out likely non-version tags. We then sort each role's versions in increasing order, and pair each two successive versions together to construct a list of version bumps. In the end, we extract 72 559 version increments involving 80 997 unique versions across 7 823 roles written by 1 891 authors. This significantly reduces the size of the dataset, filtering out over 15 000 roles that contain less than two versions, which are irrelevant for this study.

4) Syntactical 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 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.

5) Structural Differencing: The second strategy to extract difference metrics uses our structural model and differencing algorithm (Section V). We extract a structural model from the source code of each role version according to the bumps gathered in stage 3 (stage 5a). Of the 72 559 version increments collected, we remove 8 018 due to syntax errors or other parsing issues. The remaining 64 541 version bumps are 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. Then, we produce structural difference metrics by counting the number of occurrences of each change type.

V. 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 IV, to extract metrics of the difference between two role versions.

A. 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 not ideal for our purposes since it is intended for use during IaC file execution, rather than for static analysis. For example, Ansible pre-loads statically imported non-main task files, whereas dynamically included files only get parsed at runtime.

Figure 2 depicts our structural model. It is a hierarchical structure of four main types of elements, namely files, variables, blocks, and tasks, closely following the structure of a role, as described in Section III-B. 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 III-A). 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

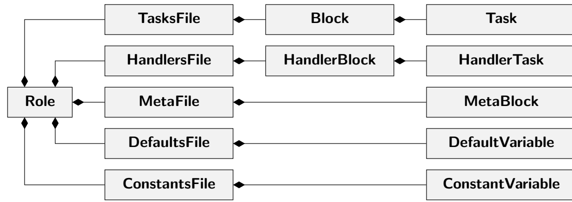

Fig. 2. The condensed UML class model of the structural role model.

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 constants files and defaults 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 III-B). 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.

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. However, this also means that roles containing syntax errors or old, now-unsupported syntax, are rejected. Such rejections account for some of the removed version pairs of stage 5 of the pipeline (cfr. Section IV).

B. 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 [16], GUMTREE [17], or CHANGENODES [18], 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.

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. These categories are summarised in Table IV. 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 added 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 IV'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 as an addition and a removal, since for a user of the role, both a rename and a removal are potentially breaking changes. As such, an edit to a variable indicates that its assigned value has changed, but its name remained unchanged.

2) Change Extraction: We will now describe our domain-specific structural difference extraction algorithm. The main challenge in extracting differences 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 structure. 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 [16], 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 the one described in [16] is that ours is specific for our structural model, which enables us to use domain knowledge to improve its results. For example, non-hierarchical structures 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

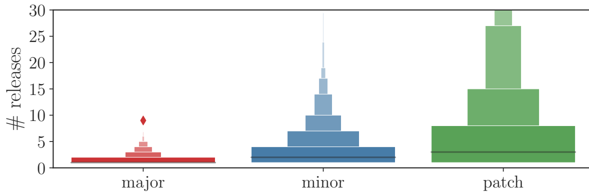

Fig. 3. Distribution of the number of Ansible role releases per release type, i.e., patch, minor, major.

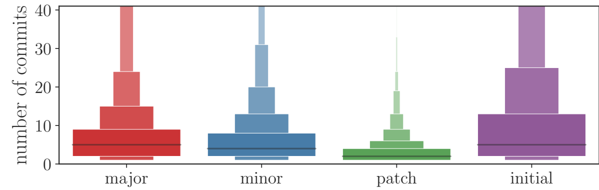

Fig. 4. Distribution of the number of commits used to release initial, patch, minor and major versions.

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.

VI. Empirical Analysis Results

The research questions in this study are organized into three parts: 1) a quantitative analysis that includes three research questions RQ_1 to RQ_3; 2) a qualitative study where we conduct surveys with Ansible role developers; and 3) the creation of a classification model to predict the suitable SemVer version bump for a new role release.

A. Quantitative Analysis

RQ_1: 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 IV) match this version. We found that 92.3% (77 640) of the version numbers use this format. The remaining, non-compliant version numbers belong to 2 008 roles, of which 1 128 roles have only 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 0.83% (701) of the versions contain such labels, belonging to 237 roles. As such, 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[3], thus, the changes may not be representative.

Next, for every two successive versions, we identify the type of version bump. 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* bump. 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 bumps per role are 3, 2, and 1, respectively.

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.

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_2: How much effort does it take to create role versions?

RQ_1 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. The mean and median number of commits are 10 and 5 for majors, 8.3 and 4 for minors, 3.5 and 2 for patches, and 14.8 and 5 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. 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 5. 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

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

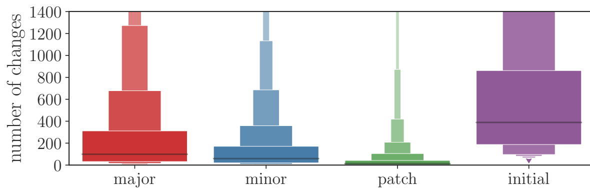

Fig. 5. Distribution of the number of lines changed (insertions + deletions) in initial, patch, minor and major version bumps.

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.

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.

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

RQ_3: Which changes are made between two role releases?

As shown in Section III, 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 syntactically edited most often. We found that only 5 of the main directories are touched in more than 10% of the releases. These 5 directories and the proportion of releases that changed them can be found in Table II. We observe that tasks are the most commonly syntactically edited component in all release types, followed by default variables. We also found that two non-standard directories have been touched in more than 10% of releases, namely `tests` (14.7%) and `molecule` (12.2%), both of which are used for testing.

Next, we investigate whether these directories are changed in isolation or are touched simultaneously with others. Figure 6 depicts a Venn diagram of touched directories for all releases. As can be seen in the figure, the majority of releases touch multiple directories, suggesting that these directories are closely related and changes to one component may trigger changes on others. Additional analysis also showed that major

TABLE II
PROPORTION OF RELEASES THAT CHANGED FILES IN THE MAIN ROLE DIRECTORIES, GROUPED BY RELEASE TYPE.

	patch	minor	major	all
tasks	48.1	73.0	78.0	55.3
defaults	29.0	56.4	67.9	37.2
meta	21.8	35.1	52.7	26.3
templates	16.1	32.5	37.5	20.9
vars	12.5	24.1	28.1	16.0

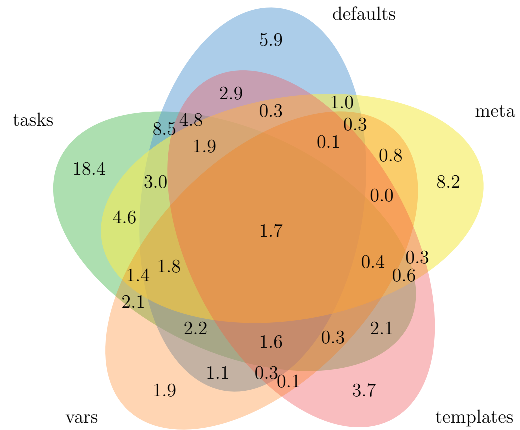

Fig. 6. Proportion of Ansible role releases that changed files in combinations of the most commonly touched directories.

releases more often touch multiple directories, whereas 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 in across the role, as was observed in RQ_2.

Up to now, we have solely considered syntactical differences, which may include semantically-irrelevant changes such as refactorings. Using the changes extracted by our structural differencing algorithm (cfr. Section V), we identify the role element types that are changed most often in specific bump types. Table III lists structural element types that have been changed (added, removed, relocated, or edited) in more than 5% of all 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 the default variables occur most frequently, in over half of the major releases and over a fourth of all releases, whereas tasks are the second most frequently changed element. Note that the numbers in this table are all lower than the numbers provided in Table II, showing that there exist releases which syntactically change those components without incurring structural changes. Surprisingly, this also leads to default variables, rather than tasks, being the most commonly structurally changed component. We therefore looked further into these changes and found that the most commonly performed changes for tasks are additions (18.07% of all releases) and relocations (17.56%), whereas for defaults, they are additions (16.12%) and value edits (14.27%). Finally, for every release type, tasks are more often removed than variables.

We also found that 37.27% of releases perform no structural change at all. This can be attributed to releases containing purely syntactical refactorings, or releases that perform changes in components 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

TABLE III
PROPORTION OF RELEASES THAT CHANGED A SPECIFIC TYPE OF ELEMENT, GROUPED BY RELEASE TYPE.

	patch	minor	major	all
Default var.	20.93	44.17	52.63	27.46
Task	18.62	42.19	44.87	24.96
Misc. metadata	10.03	18.85	32.54	13.01
Constant var.	8.99	18.36	20.79	11.58
Block	5.68	20.58	28.44	9.99
Platforms	5.24	12.88	19.95	7.59
Tasks file	2.99	12.53	17.33	5.73

line with previous observations in RQ_2.

Findings: Major releases more often change multiple role components, whereas patches more often change a single component. Tasks and default variables are changed most often. Syntactical changes often incur no structural changes.

B. Qualitative Study

We contacted a number of popular Ansible role developers to relate our findings. We selected the most downloaded roles on Galaxy, gathered their authors, and retained only those that had a publicly-listed e-mail address, of which there were 22. We then contacted each of these 22 developers and asked them the following two questions.

1) *Do you follow SemVer specifications when you change your role version number?*
2) *If so, which changes trigger you to do a patch, minor or major release?*

We received a reply from six of the 22 contacted developers. Of these six, four confirmed adherence to SemVer, one confirmed but noted that they were not strict about it, the final developer stated they use the scheme, but not the semantics. All five developers that use SemVer state that they release major versions in case of backwards-incompatible changes. However, one developer admitted to not always following this rule, e.g., in case the incompatibility is easily resolved. Four of the five developers use minor versions to introduce new functionality, whereas one claims to use it for internal, non-breaking changes such as code optimisations and refactoring. All five developers agree that patch versions should be used for bug fixes and small changes that do not affect the user.

We further asked the developers that follow SemVer what a backwards-incompatible change to the interface of an Ansible role could mean. They all appear to define the API of a role as the set of default variables and their values, and claim that removing or renaming variables leads to breaking changes in the API. One developer also mentions that they consider removing support for an older operating system or application version as a breaking change.

Findings: All surveyed developers use the SemVer version format, but some do not follow strict rules. Developers define the API of a role as its default variables, and consider removing or renaming variables to be breaking changes.

C. 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 6 developers, and our dataset contains over $1\,700$, it is impossible to generalise their answers. Moreover, as RQ_3 shows, releases often change multiple components 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 V. The dataset thus consists of the 41 structural difference features. Our approach to selecting, training, and evaluating our classifier follows the one proposed by Yan et al. [19], who identify features relevant for predicting commits that will be reverted.

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. Therefore, we focus on classification algorithms with interpretable results, so that we can inspect the decision of the resulting model. In particular, we focus on the Random Forest classifier, proposed by Ho [20], 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 [21], 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.

2) Feature Engineering: Feature engineering [22] 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 [23], where we discard correlated features to avoid collinearity. Using a threshold of 0.8, we find that the addition of handler blocks and the addition of handler files are highly correlated (0.86). As such, we discard the addition of handler files, since intuitively, adding a new file in the `handlers` directory often means a new block will be added to this file too, whereas a block can be added to an already-existing file.

Then, we perform an ablation study, where we compute the impact of features on the accuracy of the model and use this to discriminate relevant features from irrelevant ones. We use Recursive Feature Elimination (RFE) [24] 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 [25], since this works particularly well on imbalanced datasets such as ours. RFECV iteratively tests,

TABLE IV
OVERVIEW OF STRUCTURAL CHANGE CATEGORIES AND THE RANK AND WEIGHT (IN PARENTHESES) OF THEIR CORRESPONDING FEATURES IN THE CLASSIFICATION MODEL.

	Addition	Removal	Edit	Reloc.
Dependencies	13 (2.17E-3)	12 (2.71E-3)		
Platforms	**1** (3.78E-2)	2 (1.12E-2)		
Misc. metadata			**1** (5.13E-2)	
Default var.	**1** (1.81E-2)	**1** (4.42E-2)	**1** (2.83E-2)	19 (3.19E-4)
Constant var.	**1** (3.50E-2)	**1** (1.62E-2)	**1** (1.27E-2)	10 (2.93E-3)
Default var. file	8 (3.94E-3)	14 (1.95E-3)		21 (5.00E-5)
Constant var. file	**1** (1.33E-2)	3 (7.62E-3)		7 (4.67E-3)
Task	**1** (1.38E-1)	**1** (2.98E-2)	**1** (1.05E-1)	**1** (1.02E-1)
Handler task	6 (5.50E-3)	17 (9.80E-4)	**1** (1.40E-2)	15 (1.86E-3)
Block	**1** (6.64E-2)	**1** (2.56E-2)	9 (3.06E-3)	5 (6.16E-3)
Handler block	11 (2.77E-3)	18 (6.38E-4)	23 (2.85E-6)	22 (5.47E-6)
Tasks file	**1** (3.21E-2)	4 (6.93E-3)		16 (1.26E-3)
Handlers file	N/A[1]	20 (2.81E-4)		24 (1.74E-6)

[1] Valid change type, but removed through correlation analysis.

using cross-validation, which number of features maximises a specific metric, which in our case is the accuracy of the model, 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 IV. Features (i.e., change categories) with a ranking of 1 are part of the final selection, the remaining features are ranked according to their relevance, with higher ranked features being more relevant.

Of the 41 features in the dataset, 17 were selected as relevant. The majority of this selection (7) are addition changes, followed by edits (5). We further identify tasks as important elements to predict version bumps, since all change categories related to tasks are selected. Furthermore, default and constant variables are also important, with most of their changes selected as relevant. We also find that certain element types, such as handlers files and default variable files, are insignificant for the prediction, as all their features are removed.

Lastly, from each of the foldings constructed by cross-validation, we extract the weights of the trained models and compute the median weight of each feature. This provides us with a metric to assess the absolute relevance of each feature, which is also depicted in Table IV. Here, four features stand out in relevance. The addition of default variables is the best ranked feature with a score of 0.181, followed by task additions (0.138), edits (0.105), and relocations (0.102).

3) Training and Evaluation of the Final Classifier: Using the 17 features retained from the previous step, we train a Random Forest classifier to predict the bump type for each of the version increments in the dataset. We then evaluate this classifier by having it predict the class for each of these increments. Table V 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 0.8, 0.64, and 0.83 for patch, minor, and major releases, respectively. The accompanying recall scores are 0.97, 0.28, and 0.12 respectively. As such, when the release in actuality is a patch, the classifier will likely predict it as such. Moreover, when it predicts a major version, it is likely correct. This is offset by the low

TABLE V
CONFUSION MATRIX RESULT FROM THE EVALUATION OF THE MODEL.

	patch*	minor*	major*
patch	46 223	1 472	31
minor	9 894	3 866	40
major	1 974	684	357

recall for major, meaning that although a major prediction is likely correct, it is an under-approximation. Furthermore, minor releases show a relatively low precision and recall, and the confusion matrix suggests that the classifier often fails to distinguish between minors and patches. This could suggest that the distinction between these two bump types is rather vague, or alternatively, that structural features are insufficient to uncover this distinction.

Findings: The majority of structural features are irrelevant to distinguish the three types of version bumps. Additions of default variables, task additions, task edits, and task relocations 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.

VII. DISCUSSION

Our findings in RQ_1 show that most Ansible role developers follow the SemVer format. Moreover, the results of RQ_2 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 [26]. 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.

From the qualitative study, we can extract some general rules to distinguish the SemVer bump types for Ansible roles. Major bumps containing breaking changes appear to incur some reduction in the flexibility of the client, e.g., by removing variables so that clients can customise less of the role's behaviour. On the contrary, minor releases provide new functionality, often in the form of providing the client with more flexibility, e.g., by adding new variables to customise more of the role's behaviour. However, these are likely over-simplifications, as a version bump is rarely limited to changes in one component (RQ_3) and not all developers necessarily follow such rules consistently. Nonetheless, our classifier's

selection of features 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 of platforms. The removal of a platform, although not selected as a feature, is still important, as it is the highest-ranked feature among the discarded ones. This suggests that in addition to default variables, the platforms that the role support form another part of its interface. These features also coincide with the most common changes to the role (cfr. Table III).

Since developers may not all follow the same rules, our dataset may contain incorrectly tagged versions. As such, our classifier may pick up on incorrect rules. Furthermore, structural changes may not always be sufficient to distinguish release types. We therefore manually investigate a sample of its misclassifications, applying the rules extracted from the survey. More specifically, we sample 20 cases where a patch was predicted while the actual bump type is a minor, and 20 cases where a minor bump was predicted whereas the actual bump was a patch. Further, we sampled 20 false positives and 20 false negatives of major releases.

Of the 20 bumps that were incorrectly predicted to be a major version, we find that 15 contain a reduction in flexibility, e.g., the removal of variables. However, of the 20 major bumps that the classifier failed to identify, we find 9 that contain a breaking change. Most of these are removals of platforms, which was not selected as a feature, and incrementing the minimum Ansible version required to run the role, which is not modelled as a feature. Of the 20 minor bumps that were classified as patches, we found 16 that add new functionality which our classifier did not recognise. However, of the 20 patch bumps that were classified as minors, 11 contained additions of platforms or variables, suggesting they should have been minor versions. This strongly suggests that the distinction between patches and minors is not consistently applied across authors, and as such, our classifier cannot infer accurate rules.

Note that some breaking changes can be more severe than others, requiring more effort from a client to resolve. In future work, we plan to involve the clients of a role, and estimate the breakage potential of major bumps and their costs for clients.

A. Potential 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 [27].

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.

As a threat to *internal validity*, we did not consider all version bumps of Ansible roles, since we removed tagged version numbers (e.g., pre-release 1.0.0-alpha1). This may have partially influenced our results. However, we are interested in stable versions only, as such, this filtering does not affect our findings. Moreover, as discussed previously, the gathered dataset is not the ground truth. To alleviate this issue, we used feature elimination and majority voting to be more resistant to noise. In addition, we manually inspected a sample of our classifier's misclassifications to better understand the effects. 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.

As a threat to external validity, we cannot claim that our findings generalise to Infrastructure-as-Code projects for platforms other than Ansible.

VIII. 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 7 000 roles and over 80 000 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 syntactical changes between releases do not incur a structural change, and that many role releases change multiple role components. 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 6 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 flexibility for the client should lead to a new minor version, and conversely, that the removal of such flexibility is a breaking change that should lead to a major increment. However, although many developers appear to follow such guidelines implicitly, they do not do so consistently.

Acknowledgements

This research was partially funded by the Excellence of Science project 30446992 SECO-Assist financed by FWO-Vlaanderen and F.R.S.-FNRS.

References

[1] Michele Guerriero, Martin Garriga, Damian A. Tamburri, and Fabio 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.

[2] Akond Rahman, Chris Parnin, and Laurie Williams. The seven sins: Security smells in infrastructure as code scripts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE19)*, pages 164–175, 2019.

[3] E. Van der Bent, J. Hage, J. Visser, and 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.

[4] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. Toward a catalog of software quality metrics for infrastructure code. *Journal of Systems and Software*, 170:110726, 2020.

[5] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams. A systematic mapping study of infrastructure as code research. *Information and Software Technology*, 108, 2019.

[6] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Testing idempotence for infrastructure as code. In *Proceedings of the 14th International Middleware Conference (Middleware13)*, 2013.

[7] R. Shambaugh, A. Weiss, and 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.

[8] K. Ikeshita, F. Ishikawa, and 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.

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

[10] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR16)*, pages 189–200, 2016.

[11] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM14)*, 2014.

[12] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian 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)*, pages 109–120. ACM, 2016.

[13] Christopher Bogart, Anna Filippova, Christian Kästner, and James Herbsleb. Survey of ecosystem values. http://breakingapis.org/survey/. accessed: 28/10/2017.

[14] Alexandre Decan and Tom Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 2019.

[15] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. Dependency versioning in the wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 349–359. IEEE, 2019.

[16] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, November 2007.

[17] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE14)*, 2014.

[18] Reinout Stevens and Coen De Roover. Extracting executable transformations from distilled code changes. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER17)*, 2017.

[19] Meng Yan, Xin Xia, David Lo, Ahmed E Hassan, and Shanping Li. Characterizing and identifying reverted commits. *Empirical Software Engineering*, 24(4):2171–2208, 2019.

[20] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.

[21] Tin Kam Ho. Random decision forests. In *Proceedings of the 3rd International Conference on Document Analysis and Recognition (ICDAR95)*, pages 278–282, 1995.

[22] Alice Zheng and Amanda Casari. *Feature engineering for machine learning: principles and techniques for data scientists*. O'Reilly Media, Inc., 2018.

[23] M Hazewinkel. Correlation in statistics. In *Encyclopedia of Mathematics*. Kluwer, 2001.

[24] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Machine Learning*, 46(1-3):389–422, 2002.

[25] Konstantinos Sechidis, Grigorios Tsoumakas, and Ioannis Vlahavas. On the stratification of multi-label data. *Machine Learning and Knowledge Discovery in Databases*, pages 145–158, 2011.

[26] Ahmed Zerouali, Tom Mens, Jesus Gonzalez-Barahona, Alexandre Decan, Eleni Constantinou, and Gregorio Robles. A formal framework for measuring technical lag in component repositories—and its application to npm. *Journal of Software: Evolution and Process*, 31(8):e2157, 2019.

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