An Empirical Study of Utilization of Imperative Modules in Ansible

Shoma Kokuryo*, Masanari Kondo*, Osamu Mizuno*
*Software Engineering Laboratory (SEL), Kyoto Institute of Technology, Japan
{s-kokuryo, m-kondo}@se.is.kit.ac.jp, o-mizuno@kit.ac.jp

***Abstract*—In recent years, a configuration management tool is adopted to manage complicated and huge systems such as bare-metal servers, cloud computing resources and our personal computers. Such a tool makes the operations to deploy services more efficient and eliminates dependencies on the specific system operators. The operations are required to be idempotent for reproducible deployment. However, the *imperative modules* whose operations may not be idempotent are used frequently to execute user-defined scripts on the target system; it is unclear why and how they are used, though using them frequently is believed to be a bad practice.**

In this paper, we studied *why* and *how* imperative modules are used in a configuration management tool, Ansible. We found that imperative modules are mainly used to perform operations that are not supported by Ansible, and about 45% of imperative modules are replaceable by other modules; the replaceable modules might be idempotent. We, therefore, recommend developers to look at replaceable modules before using imperative modules since replaceable modules might make their operations idempotent.

***Index Terms*—Ansible; idempotency; imperative modules; infrastructure as code; configuration management;**

I. Introduction

DevOps is a software engineering paradigm, which intends to reduce the technical/organizational gap between software developers and operators [2], [4], [22]. Such a paradigm is more important in recent years since software developments entail performing bug-fix and enhancement promptly.

One of the practices in DevOps for managing infrastructures using configuration files is known as Infrastructure as Code (IaC) [2]. IaC contributes to efficient management for complex and huge operations [13]. We can use various *configuration tools* to implement IaC (e. g., Ansible [15], Puppet [14] and Chef [5]). For such configuration tools, the reproducibility of their automation scripts is important. We believe that the automation scripts yield the same result even if we execute them multiple times; this concept is called *idempotency* [6]. An idempotent operation results in the same state even if we apply the operation multiple times. Such configuration tools provide developers with *modules* that could be used to implement idempotent operations and modules that could result in non-idempotent operations such as *imperative modules*. Although developers regard using imperative modules as a bad practice (e. g., break the reproducibility) [21], [23], such modules are frequently used in practice [11]. However, to the best of our knowledge, nobody studies why and how developers use imperative modules so far.

In this paper, we studied why and how developers use imperative modules in practice in a popular configuration tool, Ansible. More specifically, we investigated four imperative Ansible *modules*: `command`, `shell`, `raw` and `script` that are used in Ansible *tasks* for each Ansible *role*. A module indicates the minimum operation; a task executes an operation using modules; a role indicates a sequence of tasks. We conducted experiments on the dataset that was retrieved from publicly available web sites: Ansible Galaxy and GitHub. In particular, we focus on the following research questions:

RQ1: **How often do developers use imperative modules?**
RQ2: **What operations do developers execute using imperative modules?**
RQ3: **How do developers keep their operations idempotent?**
RQ4: **Can we replace tasks with functionals in Ansible?**

For RQ1, we found about 49.3% of the Ansible roles use at least one of the imperative modules. For RQ2, we found developers use the imperative modules for application-specific operations that are not supported by Ansible. For RQ3, we found developers use the conditionals of Ansible tasks. For RQ4, we found about 45.0% of Ansible tasks can be replaced with other functionals.

Our results imply that we could relieve a challenge that is using imperative modules in Ansible. In addition, we could propose a recommendation tool that suggests replaceable functionals for imperative modules based on our results.

The organization of our paper is as follows. Section II describes key materials surrounding IaC and Ansible. Section III presents our experimental dataset. Section IV presents the results of our experiment. Section V discusses these results. Section VI contextualize our research in the IaC domain. Section VII describes the threats to the validity of our findings. Section VIII presents the conclusion.

II. Background

A. Infrastructure as Code (IaC)

IaC is one of the important characteristics of DevOps. To deal with the faster release cycle, developers need to continuously improve their environments to make their development cycle faster. IaC is an idea to address this challenge.

To address this challenge, IaC introduces the concept of source code to the configuration of environments. Each of the source code indicates an environment and we can build the

978-1-7281-8913-0/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS51102.2020.00063

environment according to the source code. The source code is a text file; and therefore, we can apply the best practice in source code such as version control.

B. Idempotency

An idempotent operation results in the same state even if we apply the operation multiple times [6]. For example, the multiplication of an arbitrary number by zero and the evaluation of the absolute value are idempotent.

C. Configuration Management Tool

A configuration management tool is a tool to implement IaC. Such a tool is mainly used to build the environments of server machines and virtual machines. Ansible, Chef, and Puppet are frequently used as configuration management tools. The main purpose of this tool is to automatically build an environment with idempotency. In this paper, we focus on Ansible to deeply look into it.

D. Ansible

Ansible is a configuration management tool we study in this paper, and an OSS developed by RedHat, Inc.

We describe the important terminologies:

- *Module:* The minimum unit in Ansible. A module corresponds to an operation. For example, making a new directory in a machine could be operated by a module. The number of modules is 3,387 in Ansible 2.9.4.
- *Task:* A unit to execute an operation with a sequence of modules.
- *Role:* A unit to reuse a sequence of tasks. For example, adding an admin user and changing access permission could be defined as a role. A role can be defined by a certain file structure [17].
- *Playbook:* A unit to write a configuration of a target machine based on Ansible tasks and roles.

We can use dynamic expressions and access to variables with Jinja2 templating [12] in Ansible. We can also use conditional branches and loops.

E. Declarative and Imperative Implementations

Ansible modules can be separated into two types: *declarative* and *imperative* modules. Declarative modules declare a state of a target machine and the modules decide the procedure to be the state. Imperative modules execute operations that are written in an Ansible playbook.

Declarative modules ensure that the operations are idempotent themselves while users need to ensure idempotency for imperative modules. Since it is difficult to ensure idempotency for imperative modules, such imperative modules might induce problems [9], [11], [21], [23].

In this paper, we study imperative modules in order to reveal why and how are imperative modules used that might be non-idempotent. Since Ansible does not publicly distinguish the imperative modules and declarative modules, we regard four modules that can execute user scripts as imperative modules that are frequently used in practice and are backwardly compatible: `command`, `shell`, `raw`, and `script`.

F. Results of Ansible Tasks

All tasks return a value to show the result. Let me assume that a task consists of idempotent modules and such a task is applied to a target machine C. If we have no errors and C meets all conditionals of the task, the task would return `changed` or `ok` in the first time execution; however, the task would return `ok` in the second and after executions.

III. Dataset

In this section, we discuss our studied datasets.

A. Ansible Galaxy

Ansible Galaxy[1] is a publicly available web site to search, download and share Ansible roles [16]. All Ansible roles on Ansible Galaxy are linked with repositories on GitHub; we can get the links via the Web API.

B. Dataset Extraction

We retrieve the list of Ansible roles by using the Web API of Ansible Galaxy with `galaxy_crawler`[2]. We use the list to retrieve the source code from GitHub by using `git clone` command. We select Ansible roles according to the following conditions:

- Ansible roles that were updated within the past year.
- Ansible roles that are in the top-10 percentile in terms of number of downloads.

C. Overview of Studied Datasets

We extracted 21,967 Ansible roles by the Web API and cloned the associated repositories from GitHub on Oct. 22, 2019. The number of Ansible roles that were updated at least one time between Oct. 22, 2018 and Oct. 22, 2019 is 8,832. We retrieved 884 Ansible roles that are in the top-10 percentile in terms of the number of downloads from 8,832.

We cloned 865 repositories from GitHub by using the list of 884 Ansible roles. This is because one repository can store multiple Ansible roles. In addition, we cannot retrieve one repository[3]. Hence, we finally get 864 repositories.

IV. Experimental Results

In this section, we show the results. We discuss the results in Section V.

A. RQ1: How often do developers use imperative modules?

Motivation: Developers frequently use the imperative modules in practice [11]. In this RQ, we clarify the number of appearances of imperative modules in Ansible.

Approach: To extract all tasks from the dataset, we first extract all Ansible roles. A role consists of multiple tasks and such tasks are defined as YAML files. We parse all YAML files following the specification of YAML 1.1[4] [1]. In addition, Ansible has `block` that enables developers to group multiple

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

[2] https://github.com/pddg/galaxy_crawler

[3] The reason might be a move, delete or close the repository.

[4] We do not substitute variables (e. g., "{{bar}}" would not be extracted)

TABLE I
THE PROPORTION OF USES OF THE IMPERATIVE MODULES.

Module	# Appearances	Proportion (%)
command	1,608	60.54
shell	905	34.07
raw	114	4.29
script	29	1.09

TABLE II
THE PROPORTION OF ROLES THAT USE EACH OF THE IMPERATIVE MODULES. NOTE THAT WE CAN USE TWO OR MORE DIFFERENT IMPERATIVE MODULES ON A ROLE; AND THEREFORE, THE SUM OF FOUR NUMBERS/PROPORTIONS OF ROLES AND THE TOTAL NUMBER/PROPORTION ARE DIFFERENT.

Module	# Roles	Proportion of Roles (%)
command	267	31.49
shell	171	20.16
raw	15	1.77
script	7	0.83
Total	418	49.29

TABLE III
THE PROPORTION OF THE IMPERATIVE MODULES IN THE TOP-5 ROLES IN TERMS OF THE NUMBER OF USES.

Role	# Imperative Modules	Proportion (%)
MindPointGroup.RHEL7-CIS	471	17.73
florianutz.Ubuntu1604-CIS	99	3.73
oVirt.hosted_engine_setup	93	3.50
anthcourtney.cis-amazon-linux	79	2.97
ceph.ceph_handler	44	1.65

tasks. Since the `block` can be nested, we recursively search for the inside of the `block` in order to extract all tasks. We identify modules for each extracted task and count the number of appearances of modules.

Results: We successfully extract tasks from 848 out of 884 Ansible roles, which include 19,542 tasks. The proportion of tasks that uses at least one imperative module is about 13.59% (2,656/19,542). Table I shows the proportion of uses of the imperative modules.

The proportion of Ansible roles that uses at least one imperative module is about 49.29%. Table II shows the proportion of each imperative module.

The utilization of the imperative modules is imbalance across the roles. Table III shows the proportion of the imperative modules for the roles in the top-5 number of uses. We observe that the proportion of the imperative modules in MindPointGroup.RHEL7-CIS is about 17.73%; however, the others are less than 4%.

B. RQ2: What operations do developers execute using imperative modules?

Motivation: Hummer et al. [11] showed that the imperative modules are frequently used. However, to the best of our knowledge, nobody investigates which operations are the imperative modules used. In this RQ, we study the operations with the imperative modules.

TABLE IV
OVERVIEW OF ANSIBLE TASK CATEGORIES IN OUR EXPERIMENTS.

Category	Definition	Example
app	Application-specific operation	a2enmod foo pyenv install foo
file	File operation	touch foo.txt grep foo var.txt
system	OS configuration and service operation	systemctl enable foo sysctl -n foo
package	Software package operation	apt-get install foo pip install foo
script	Script execution	./foo.sh python foo.py
version	Version check operation	python -V mysql –version
db	Database operation	mysql -e “foo” psql -c “foo”
fs	File system operation	mount foo var resize2fs
which	Callable command check operation	which python type mysql
network	Network operation	curl foo wget foo
build	Build operation	./configure make
selinux	SELinux operation	getenforce semodule -i foo
validation	Configuration file verification	nginx -t visudo -cf foo
other	Others	date, sleep foo

Approach: We manually classify the Ansible tasks that were extracted in RQ1 into the 14 categories in Table IV. We defined 14 categories that are frequently used based on our investigation on the tasks. Since we also need to read the surrounding lines of a target task, we manually read an entire file. If a task consists of multiple category commands, we decide a category based on the purpose of all commands.

In addition, we classify the tasks into three access-types to study the difference of tasks that change status or not: “read-only,” “read and write,” and “unknown.” More specifically, if any commands of a task are difficult to understand due to variables or unknown operations such as script execution, we classify the task into “unknown.” If all commands of a task are read-only, we classify such a task into “read-only.” Otherwise, we classify such a task into “read and write.”

One of the authors who has been using Ansible for three years in order to manage server computers in a laboratory conducted this manual analysis. To verify the result, the author conducted the manual analysis twice.

Results: The first most frequently used operation is app (24.26%). The second and third most frequently used operations are file (17.51%) and system (15.67%). These top-three categories account for over half of the tasks (57.44%). Table V shows the proportion of the 14 task categories.

We observe that app has many read and write tasks while file has many read-only tasks. Table VI shows the number of appearances of tasks for each category and access-type.

TABLE V
THE PROPORTION OF TASKS FOR THE 14 TASK CATEGORIES.

Category	# Appearances	Proportion
app	644	24.26
file	465	17.51
system	416	15.67
package	311	11.71
script	151	5.69
version	111	4.18
db	94	3.54
fs	75	2.83
which	56	2.11
network	44	1.66
build	43	1.62
selinux	25	0.94
validation	8	0.30
other	212	7.98

TABLE VI
THE NUMBER OF APPEARANCES OF TASKS FOR EACH CATEGORY. WE CLASSIFY ALL THE TASKS FOR EACH CATEGORY INTO THREE ACCESS-TYPES: "READ-ONLY" (R), "READ AND WRITE" (W), AND "UNKNOWN" (U).

Category	Access-Type		
	R	W	U
app	128	489	27
file	322	143	0
system	237	178	1
package	108	201	2
other	143	6	63
script	5	18	128
version	109	1	1
db	17	70	7
fs	31	43	1
which	55	1	0
network	17	27	0
build	0	42	1
selinux	9	16	0
validation	8	0	0

C. RQ3: How do developers keep their operations idempotent?

Motivation: To implement the idempotent operations with the imperative modules, developers need to keep their operations idempotent. The conditionals help developers to achieve such a purpose. Hence, in this RQ, we study how do developers use such conditionals in order to keep their operations idempotent.

Approach: Ansible tasks can be defined with conditionals for its execution. Three of the imperative modules, `command`, `shell`, and `script` have three conditionals: `when`, `creates`, and `removes`; one of the imperative modules, `raw` has a conditional: `when`. `when` is a common conditional across all Ansible modules[5]. `creates` and `removes` are the specific conditionals for the three imperative modules; they decide the module runs based on if the designated path exists. We study these three conditionals in this RQ.

[5]https://docs.ansible.com/ansible/2.9/user_guide/playbooks_conditionals.html

TABLE VII
THE PROPORTION OF TASKS WITH CONDITIONALS.

	Imperative Modules				Others	Total
	R	W	U	Total		
Yes	360	862	141	1,363	7,896	9,259
No	829	373	91	1,293	8,990	10,283
Proportion (%)	30.28	69.80	60.78	51.32	46.76	47.38

If developers use a `block`, multiple tasks are grouped and we can use `when` for the entire `block`. Hence, we parse all `block`s and consider `when` for the `block`s as well.

Although the imperative modules always result in `changed`, we can override the result by using `changed_when`[6] to avoid executing redundant tasks. For example, if we use the imperative modules, we would always execute the tasks in the `handlers` directory; however, using `changed_when` could avoid the redundant execution when the imperative modules do not change certain conditionals that are described in `changed_when`.

Results: **The tasks with any imperative modules use conditionals more frequently rather than the other tasks.** Table VII shows the proportion of tasks with conditionals. The proportion of the tasks that use any conditionals and any imperative modules is 51.32% while the proportion of the other tasks that use any conditionals is 46.76%. Interestingly, the proportion of the read and write tasks (W) that use any conditionals and any imperative modules is 69.80% while the proportion of the read-only tasks (R) is 30.28%. Hence, the read and write tasks (W) with any imperative modules use conditionals more frequently rather than the read-only tasks (R).

The tasks with any imperative modules use `changed_when` more frequently rather than the other tasks. Table VIII shows the proportion of tasks with `changed_when`. The proportion of the tasks with the imperative modules that use `changed_when` is 49.57% higher compared to the other tasks (50.75% - 1.18%). In addition, the proportion of the read-only tasks that use `changed_when` is 83.85% while the read and write tasks is only 23.80%.

The proportion of false in the table indicates the proportion of the conditional of `changed_when` is false. Interestingly, Almost all read-only tasks (97.99%) use this conditional; the total proportion is also high (91.99%). This result shows that developers set the conditional of "not changed" for almost all tasks that use `changed_when` regardless of their execution.

D. RQ4: Can we replace tasks with functionals in Ansible?

Motivation: Ideally, we should avoid using the imperative modules to keep our IaC scripts idempotent. However, we might have the case where we need to use the imperative

[6]https://docs.ansible.com/ansible/2.9/user_guide/playbooks_error_handling.html

TABLE VIII
THE PROPORTION OF TASKS WITH changed_when.

	Imperative Modules				Others	Total
	R	W	U	Total		
Yes	192	941	175	1,308	16,686	17,994
No	997	294	57	1,348	200	1,548
Proportion (%)	83.85	23.80	24.57	50.75	1.18	7.92
Proportion of False (%)	97.99	78.91	77.19	92.95	85.50	91.99

TABLE IX
THE PROPORTION OF REPLACEABLE TASKS FOR EACH ACCESS-TYPE.

Access-Type	# Replaceable	# Non-Replaceable	Proportion (%)
R	689	500	57.95
W	505	730	40.89
U	2	230	0.86
Total	1,196	1,460	45.03

modules. In this RQ, we study replaceable tasks in order to clarify why developers use the imperative modules.

Approach: We execute the same procedure with RQ2; we manually look at the tasks. We search for *replaceable functionals* that can be used to replace the tasks, regarding not only modules but also Ansible variables (*Facts*) and Ansible plugins (e. g., *Filters*)[7] as the *functionals*. We define *replaceable tasks* as follows:

- Tasks that can be replaced with functionals in Ansible. Any combinations of functionals are acceptable.
- The results and operations are the same between before and after replacing tasks except for the output formats.
- The side effect is acceptable if such effect is the read-only operation.
- The entire task can be replaceable.

We show examples. Listing 1 shows a task to install `wget` with `apt-get`. This task can be replaced with `apt` module. Listing 2 shows a task where we use the `apt` module instead. Listing 3 shows a task where we can use Facts. Facts retrieve information from our remote systems when we connect them. The information can be available in a task. In fact, we do not need to execute any operations to retrieve information because of Facts; we show an example in Listing 4.

For each task, we investigate the replaceability. If such a task is not replaceable, we study the reason.

Results: **The tasks that are classified into R are more replaceable rather than those that are classified into W.** 45.03% of the tasks are replaceable with the functionals. The proportion of the read-only tasks (R) that can be replaceable is 57.95% (689) while the proportion of the read and write tasks (W) is 40.89% (505). Table IX shows the proportion of replaceable tasks for each access-type.

A few application-specific operations (app) are replaceable while file operations (file) and OS configuration and service operation (system) are more replaceable. The

[7]https://docs.ansible.com/ansible/2.9/plugins/plugins.html

```
---
- name: Install wget by apt-get
  command: apt-get install wget
```

Listing 1. apt module can be used to replace.

```
- name: Install wget by apt-get
  apt:
    name:
      - wget
```

Listing 2. Listing 1 that is replaced with the apt module.

```
- name: Get Kernel release
  command: uname -r
  register: KERNEL_RELEASE
  changed_when: false
```

Listing 3. Facts can be used to replace [8]. The original file can be found at https://github.com/anthcourtney/ansible-role-cis-amazon-linux/blob/v1.1.8-beta/tasks/level-1/4.2.4.yml.

```
# we do not need this module in practice
- debug:
    msg: "{{ ansible_kernel }}"
```

Listing 4. Listing 3 that is replaced with the Facts.

TABLE X
THE PROPORTION OF REPLACEABLE TASKS FOR EACH MODULE CATEGORY (DESCENDING ORDER OF # TASKS).

Category	# Replaceable	# Non-Replaceable	Proportion (%)
app	149	495	23.14
file	393	72	84.57
system	234	182	56.25
package	163	148	52.41
script	0	151	0.00
version	6	105	5.405
db	46	48	48.94
fs	31	44	41.33
which	0	56	0.00
network	16	28	36.37
build	20	23	46.51
selinux	1	24	4.00
validation	0	8	0.00
other	137	75	64.62

first most frequently used category (app) includes the largest number of tasks while the proportion of replaceable tasks is low (23.14%). The proportions of the second and third most frequently used categories are high (84.57% and 56.25%). Table X shows the proportion of replaceable tasks for each module category.

The proportions of tasks that can be replaced with functionals that correspond to popular UNIX commands or Facts are high. `file` has the highest proportion in terms of the number of replaceable tasks. `service_facts`, `package_facts`, and `facts` account for about 20% of replaceable tasks. Table XI shows the proportion of replaceable tasks for each module/plugin. Note that we count the related modules as one module. For example, we regard `file` module and `file` plugin as `file`; modules that are related to OpenSSL are regarded as `openssl`. `debug` includes unnecessary tasks. `facts` indicate tasks that can be replaced with Facts.

Unsupported operations are the most common reason

TABLE XI
THE PROPORTION OF REPLACEABLE TASKS FOR EACH MODULE/PLUGIN THAT ARE IN THE TOP-10 IN TERMS OF THE PROPORTION.

Module/Plugin	# Appearances	Proportion (%)
file	129	10.79
debug	126	10.54
service_facts	108	9.03
package_facts	81	6.77
find	76	6.35
stat	65	5.43
facts	54	4.52
copy	53	4.43
openssl	45	3.76
systemd	34	2.84

TABLE XII
THE REASON FOR NON-REPLACEABILITY.

Reason	# Appearances	Proportion(%)
Unsupported operations	1,099	75.27
Unknown	225	15.27
Keep compatibility	82	5.62
To be too complex	33	2.26
Avoid defects	23	1.56

TABLE XIII
THE PROPORTION OF REPLACEABLE MODULES IN THE TOP-5 ROLES IN TERMS OF THE NUMBER OF USES OF THE IMPERATIVE MODULES.

Role	# Imperative Modules	# Replaceable	Proportion (%)
MindPointGroup.RHEL7-CIS	471	383	81.32
florianutz.Ubuntu1604-CIS	99	89	89.90
oVirt.hosted_engine_setup	93	20	21.51
anthcourtney.cis-amazon-linux	79	35	44.30
ceph.ceph_handler	44	17	38.64

why such tasks are non-replaceable. The unsupported operations account for 75.27% (Table XII). This reason is the majority of non-replaceable modules.

Over 80% of imperative modules in the top-one and -two roles in terms of the number of uses of the imperative modules are replaceable. Table XIII shows the proportion of replaceable modules in the top-5 roles in terms of the number of uses of the imperative modules. We observe that MindPointGroup.RHEL7-CIS and florianutz.Ubuntu1604-CIS have high proportions (81.32% and 89.90%).

V. DISCUSSION

A. Unsupported Modules in Ansible

Given the results of RQ1, half of the roles has tasks that use at least one imperative module (Table II). About 24% of such tasks implement the application-specific operation (RQ2: Table V). The most common reason for non-replaceable tasks is that such tasks are unsupported by Ansible (RQ4: Table XII). These results imply that developers have to use the imperative modules for managing unsupported applications by Ansible.

Using the imperative modules is unavoidable in order to implement unsupported operations. Hence, when we evaluate IaC scripts that implement an operation, we need to check whether such an operation is supported by the tool.

B. Developers and Idempotency

About 70% of the read and write tasks that use any imperative modules use conditionals such as `when` (RQ3: Table VII). The read-only tasks account for only about 30%; and therefore, the read and write tasks more frequently use conditionals. On the contrary, the read-only tasks that use `changed_when` account for about 84% while the read and write tasks account for about 24% (RQ3: Table VIII). Hence, the read-only tasks more frequently use `changed_when` that can override the result. The read-only tasks do not change anything even if such tasks use the imperative modules. The read and write tasks could change status; developers need to check the condition before the execution. This result implies that developers are aware of the difference between the read-only tasks and the read and write tasks, and intend to keep their tasks idempotent.

About 30% of the read and write tasks that use any imperative modules do not have any conditionals (RQ3: Table VII). Such tasks might not be idempotent since their execution is uncontrolled. However, there exist exceptions. For example, the changes that are induced by non-idempotent operations have no challenges in practice.

C. Imbalanced Usage of Imperative Modules

Given the RQ1 results, the tasks in the roles in the top-5 percentiles account for about 30% of all the tasks that use any imperative modules (Table III). We have already discussed the reason why developers use unsupported operations. However, the roles of MindPointGroup.RHEL7-CIS, florianutz.Ubuntu1604-CIS, and anthcourtney.cis-amazon-linux use not only many imperative modules but also many replaceable modules (Table XIII). These roles seem to be related to Center for Internet Security[8]; a part of developers might share bad practices about using modules.

The roles might not require idempotent operations and might allow non-idempotent operations (imperative modules) to induce side effects in practice. However, using the Ansible modules have advantages: scripts would be understandable and we would use the community supports. We recommend developers to use modules if such modules are applicable.

D. Bad Practices

We discuss the bad practices that we found in our study.

Execution of an external script: We found that developers use external scripts such as shell scripts. Using these scripts induces additional maintenance efforts: maintaining external scripts and keep external scripts idempotent. We should divide such scripts into a sequence of the Ansible tasks in order to keep the operations idempotent.

Imperative Use of Declarative API: All we need to use the declarative modules in Ansible is to write the desirable state of our machine. However, we found the case where the

[8]https://www.cisecurity.org/

```
---
- name: Check that wget is installed
  command: which wget
  register: wget

- name: Install wget
  apt:
    name:
      - wget
  when: wget.rc != 0
```

Listing 5. Imperative use of declarative API.

declarative modules are executed with unnecessary conditionals. For example, Listing 5 defines a desirable state; however, `when` module is used to check the current state. The declarative module covers the current status check as well. We call one that the declarative modules are used with the conditionals to control the execution as *imperative use of declarative API*.

We found two reasons to use the imperative use of declarative API: (1) The associated modules have defects. (2) Developers want to use the modules that were unimplemented during the previous version while keeping the backward compatibility.

For example, the Facts of `ansible_selinux` always returns false regardless of the state when the Python library is not installed in SELinux[9] because of a defect. Hence, developers need to use the imperative modules to check the status.

Developers Avoid Using Facts Due to Lack of Knowledge about Ansible Modules: `service_facts`, `package_facts`, and `facts` account for about 20% of replaceable tasks (Table XI). `service_facts` and `package_facts` are implemented as modules; these modules are read-only to collect the status of services and installed packages. This result implies that developers prefer common commands such as `grep` rather than Facts. We describe two possible reasons.

One of the reasons is the lack of the Ansible documents. The document about Facts [18] shows examples; however, the document does not describe the behavior of Facts in different OSs and the process of them. Developers, therefore, need to look at the behavior of Facts.

Another reason is the lack of knowledge about the modules. The top of the replaceable modules includes the modules that correspond to popular UNIX commands such as `file` and `find`. This result might imply that developers do not deeply understand the details of the modules (e. g., functionals, specifications, and behavior); and therefore, they use popular UNIX commands. In addition, Ansible includes over 3,000 modules. This gigantic number of modules might make the Ansible documentation difficult to read and seek.

VI. Related Work

A. Idempotency on IaC

Ansible developers require to ensure their scripts idempotency. We can use frameworks (e. g., Molecule [7]) to encourage such a task. Testing frameworks for idempotency on Chef and Puppet are also proposed [10], [11]. These frameworks test the convergence of the target system status by comparing changes of target system status for each execution. However, such frameworks need much processing time. We discussed in Section V-B that developers frequently use conditionals to control the execution of the read and write tasks that use the imperative modules. Hence, investigating the operations and conditionals of imperative modules might identify the idempotency on the usage of imperative modules.

Shambaugh et al. [20] proposed a tool *Rehearsal* that implements determinacy analysis with statistical analysis for Puppet. Rehearsal replaces *resources*[10] with their unique imperative language and analyze the scripts. However, Rehearsal excludes a resource `exec` from the target resources since `exec` executes user scripts that are generally difficult to replace. Although the statistical analysis is effective, Hanappi et al. [10] reported that the applicable scripts are limited in practice. Indeed, our findings show that the most frequently used operation is application-specific operations; it is difficult to apply the statistical analysis to these IaC scripts.

B. Quality Model of IaC Scripts

IaC tools bring the best practices that are applied to source code into configurations while such tools also bring disadvantages. For example, as the size of the source code increases, the complexity of the source code increases. Hence, it is important to maintain the quality of IaC scripts.

We can use *code smell* to detect the quality of source code [3], [24]. Prior works have already studied code smell in IaC scripts [19], [21]. They consider using non-declarative resources as a code smell. However, our findings and results imply that these resources might implement operations that are not supported by the tools. Hence, we would need to consider unsupported operations when proposing code smell metrics.

Prior research defined a quality model on Puppet and evaluates it with Puppet developers [23]. They carried out a survey of Puppet developers about bad practices. The Puppet developers reported that using `exec` would be a bad practice; however, some developers disagree with this assumption. Our findings and results show that using the imperative modules are inevitable. Hence, we recommend using two new metrics in the quality model that are whether (1) imperative modules can be replaced with other declarative modules, and (2) imperative modules are controlled by conditionals for idempotency.

VII. Threats to validity

A. Construct Validity

We used the comments surrounding a target task to detect the replaceable functionals in our manual analysis. Such comments were written by developers; and therefore, the accuracy of the comment contents depends on developers. This accuracy would affect our experimental results.

We selected a subset (four modules) of many existing imperative modules: `command`, `shell`, `raw`, and `script`. We carefully selected these imperative modules that might be frequently used in practice and are backwardly compatible. Future studies are necessary to investigate whether our results apply to other imperative modules.

[9]https://github.com/ansible/ansible/issues/16612

[10]The resource corresponds to the Ansible module.

B. External validity

We applied our experiments to only a configuration management tool Ansible. However, Ansible has differences with other configuration management tools. Future studies are necessary to investigate whether our results generalize to other configuration management tools.

With regard to the generalizability of our results, we applied our experiments to publicly available Ansible roles in Ansible Galaxy. Ansible Galaxy is publicly available for all Ansible developers and the authors of this paper believe that these studied roles are similar to practical roles.

We failed to extract tasks from 36 roles in RQ1. The number of failed tasks is small and the reasons are valid; and therefore, these errors are not a challenge.

C. Internal validity

One of the authors manually looked at the Ansible tasks twice. However, we need the validation of our results by other Ansible developers.

We avoid extracting substitute variables. Hence, our results might be affected by such variables. However, to interpret such variables, we need to execute all Ansible roles in all possible environments. Our findings are acceptable in practice.

VIII. Conclusion

We studied the usages of the imperative modules in the configuration management tool. We found:

- About 49% of Ansible roles use at least one imperative module. Such imperative modules are mainly used to implement unsupported operations, file operations, and service configurations.
- About 70% of the tasks that use any imperative modules and would change the status have conditionals. About 84% of the tasks that use any imperative modules and are read-only override the result to be "unchanged." Hence, developers intend to keep their tasks idempotent and are aware of the difference between the read-only tasks and the read and write tasks.
- About 45% of the tasks are replaceable with other Ansible functionals. We recommend developers to check whether there exist Ansible functionals that are applicable.

Acknowledgment

This work has been supported by JSPS KAKENHI Japan (Grant Numbers: JP19J23477).

References

[1] YAML Ain't Markup Language (YAML™) Version 1.1. [Online]. Available: https://yaml.org/spec/1.1/

[2] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri, "DevOps: Introducing infrastructure-as-code," in *Proc. of the 39th International Conference on Software Engineering Companion (ICSE-C)*. Institute of Electrical and Electronics Engineers Inc., 2017, pp. 497–498.

[3] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," in *Information and Software Technology*. Elsevier B.V., 2019, pp. 115–138.

[4] C. P. Bezemer, S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. Van Hoorn, M. Villavicencio, J. Walter, and F. Willnecker, "How is performance addressed in DevOps? A survey on industrial practices," in *Proc. of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE)*. Association for Computing Machinery, Inc, 2019, pp. 45–50.

[5] Chef Software, Inc. Chef: Deploy new code faster and more frequently. automate infrastructure and applications. [Online]. Available: https://www.chef.io/

[6] A. Couch and Y. Sun, "On the algebraic structure of convergence," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2867, pp. 28–40, 2003.

[7] J. Dewey. Molecule — molecule 2.22 documentation. [Online]. Available: https://molecule.readthedocs.io/en/stable/index.html

[8] GRyCAP. ansible-role-docker/debian.yml at 45bfc55c745f3f39f8557de98ddd3d8be9b7ee89, grycap/ansible-role-docker. [Online]. Available: https://github.com/grycap/ansible-role-docker/blob/45bfc55c745f3f39f8557de98ddd3d8be9b7ee89/tasks/Debian.yml#L1-L4

[9] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *Proc. of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 580–589.

[10] O. Hanappi, W. Hummer, and S. Dustdar, "Asserting reliable convergence for configuration management scripts," in *Proc. of the 2016 ACM SIGPLAIN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Association for Computing Machinery, 2016, pp. 328–343.

[11] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, "Testing idempotence for infrastructure as code," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8275 LNCS, pp. 368–388, 2013.

[12] Pallets. Jinja — jinja documentation (2.11.x). [Online]. Available: https://jinja.palletsprojects.com/en/2.11.x/

[13] C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, M. Stumm, S. Whitaker, and L. Williams, "The Top 10 Adages in Continuous Deployment," *IEEE Software*, vol. 34, no. 3, pp. 86–95, 2017.

[14] Puppet. Powerful infrastructure automation and delivery — puppet. [Online]. Available: https://puppet.com/

[15] Red Hat, Inc. Ansible is simple it automation. [Online]. Available: https://www.ansible.com/

[16] ——. Galaxy user guide — ansible documentation. [Online]. Available: https://docs.ansible.com/ansible/latest/galaxy/user_guide.html

[17] ——. Roles — ansible documentation. [Online]. Available: https://docs.ansible.com/ansible/2.9/user_guide/playbooks_reuse_roles.html

[18] ——. Using variables — ansible documentation. [Online]. Available: https://docs.ansible.com/ansible/2.9/user_guide/playbooks_variables.html

[19] J. Schwarz, A. Steffens, and H. Lichter, "Code smells in infrastructure as code," in *Proc. of the 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2018, pp. 220–228.

[20] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: A configuration verification tool for puppet," in *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 2016, pp. 416–430.

[21] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proc. of the 13th Working Conference on Mining Software Repositories (MSR)*. Association for Computing Machinery, Inc, 2016, pp. 189–200.

[22] J. Smeds, K. Nybom, and I. Porres, "DevOps: A definition and Perceived Adoption Impediments," in *Proc. of the 16th Agile Processes in Software Engineering and Extreme Programming*. Springer International Publishing, 2015, pp. 166–177.

[23] 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 *Proc. of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Institute of Electrical and Electronics Engineers Inc., 2018, pp. 164–174.

[24] E. Van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells," in *Proc. of the Ninth Working Conference on Reverse Engineerin (WCRE)*. IEEE Computer Society, 2002, pp. 97–106.