What Constitutes the Deployment and Run-time Configuration System? An Empirical Study on OpenStack Projects

NARJES BESSGHAIER, ETS Montreal, University of Quebec, Canada
MOHAMMED SAYAGH, ETS Montreal, University of Quebec, Canada
ALI OUNI, ETS Montreal, University of Quebec, Canada
MOHAMED WIEM MKAOUER, Rochester Institute of Technology, USA

Modern software systems are designed to be deployed in different configured environments (*e.g.*, permissions, virtual resources, network connections), and adapted at run-time to different situations (*e.g.*, memory limits, enabling/disabling features, database credentials). Such a configuration during the deployment and run-time of a software system is implemented via a set of configuration files, which together constitute what we refer to as a "configuration system". Recent research efforts investigated the evolution and maintenance of configuration files. However, they merely focused on a limited part of the configuration system (*e.g.*, specific infrastructure configuration files or Dockerfiles), and their results do not generalize to the whole configuration system. To cope with such a limitation, we aim by this paper to better capture and understand what files constitute a configuration system. To do so, we leverage an open Card Sort technique to qualitatively study 1,756 configuration files from OpenStack, a large and widely studied open-source software ecosystem. Our investigation reveals the existence of nine types of configuration files, which cover the creation of the infrastructure on top of which OpenStack will be deployed, along with other types of configuration files used to customize OpenStack after its deployment. These configuration files are interconnected while being used at different deployment stages. For instance, we observe specific configuration files used during the deployment stage to create other configuration files that are used in the run-time stage. We also observe that identifying and classifying these types of files is not straightforward, as five out of the nine types can be written in similar programming languages (*e.g.*, python and bash) as regular source code files. We also found that the same file extensions (*e.g.*, `Yaml`) can be used for different configuration types, making it difficult to identify and classify configuration files. Thus, we first leverage a machine learning model to identify configuration from non-configuration files, which achieved a median AUC of 0.91, a median Brier score of 0.12, a median precision of 0.86, and a median recall of 0.83. Thereafter, we leverage a multi-class classification model to classify configuration files based on the nine configuration types. Our multi-class classification model achieved a median weighted AUC of 0.92, a median Brier score of 0.04, a median weighted precision of 0.84, and a median weighted recall of 0.82. Our analysis also shows that with only 100 labeled configuration and non-configuration files, our model reached a median AUC higher than 0.69. Furthermore, our configuration model requires a minimum of 100 configuration files to reach a median weighted AUC higher than 0.75.

CCS Concepts: • **Software and its engineering** → *Software notations and tools*; Software configuration management and version control systems;

Additional Key Words and Phrases: Configuration files, Infrastructure-as-Code, Files classification, Machine learning models, OpenStack.

Authors' addresses: Narjes Bessghaier, narjes.bessghaier.1@ens.etsmtl.ca, ETS Montreal, University of Quebec, Montreal, QC, Canada; Mohammed Sayagh, mohammed.sayagh@etsmtl.ca, ETS Montreal, University of Quebec, Montreal, QC, Canada; Ali Ouni, ali.ouni@etsmtl.ca, ETS Montreal, University of Quebec, Montreal, QC, Canada; Mohamed Wiem Mkaouer, mwmvse@rit.edu, Rochester Institute of Technology, Rochester, NY, USA.

1049-331X/2023/7-ART $15.00
https://doi.org/10.1145/3607186

1 INTRODUCTION

Configuring a software system consists of adapting and customizing it to different situations as well as platforms [1]. For example, one can enable JavaScript on Firefox via the "`javascript.enabled`" configuration option. Similarly, one can configure which tools, resources, and libraries to install on an infrastructure of a software system. For example, one can install and configure the *"Apache"* web server for the infrastructure of a web application and configure a set of access permissions such as the resources (*e.g.*, files, URLs) that users can access.

While such software configuration becomes necessary for managing complex software systems, an incorrect configuration could lead to common and severe issues. For instance, WhatsApp, Instagram, and Facebook software applications were down for several hours due to a configuration error on October 4th, 2021 [19]. This outage disrupted nearly 3 billion users and cost an almost $7 billion loss[1]. It is worth noting that this is not the first time when Facebook faced such an outage. A similar outage occurred in 2019, lasting as long as 24 hours, due to a server configuration change[2]. Other popular software systems, such as Amazon, faced configuration issues as well. In 2017, a cloud security misconfiguration exposed more than 40k clients' passwords, which were stored on Amazon S3 [68].

These errors are due to the complexity of maintaining configuration files [65], which motivated prior work [12, 25, 27] on studying the maintenance of software configuration. For instance, Cito et al. [12] examined the maintenance and evolution of Dockerfiles. Likewise, Ibrahim et al. [25] examined the maintenance and evolution of Docker-compose files. Similarly, Jiang and Adams [27] evaluated the maintenance efforts of IaC files in OpenStack by studying their co-evolution with source, test, and build files.

However, the configuration of a software system can be made through multiple configuration tools and configuration files, among which some cannot be straightforwardly identified. For example, OpenStack uses several tools to create and configure its infrastructures, such as Puppet [81], Chef [79], and Ansible [23], use configuration files written in different formats (*e.g.*, ".rb", ".yaml", ".sh", ".erb", etc.), among which these can share format or extensions with source code files (bash or python scripts). According to Guerriero et al. [21], one of the most common Infrastructure-as-Code (IaC) issues is that dealing with the different formats of IaC tools is often obscure to most and requires specialized personnel. They also pointed out the need to have an ontology that deciphers the categories and relations to aid in reasoning how IaC tools or formats may fit together in one context. Yet, Guerriero et al. [21] focused just on IaC configuration, whereas, identifying configuration files related to the whole deployment and runtime of a software system can be more challenging. For instance, developers might need to know which configuration files they need to change, architects might need to identify the configuration files to better model the configuration system, novices might need to know the existing configurations and users might need to know which configuration files to change to deploy a software system such as OpenStack. Furthermore, Sayagh et al. [65] found that practitioners have a lack of ownership over the configuration files, allowing any developer to change such files. As a side effect, developers might not be aware and keep track of all the existing configuration files. On top of that, Sayagh et al. [65] finding is limited to runtime configuration files, suggesting that it can be more challenging to identify all the possible types of configuration files of a software project. For example, an OpenStack developer "do[es] not believe there is a fixed standard for configuration files vs. code" [3]. Another example of a user who was not able to correctly configure OpenStack before their *"elder colleague"* figured out the right file to change[4], which is interestingly a Python file. In fact, knowing the configuration tools that are used in a project (*e.g.*, Ansible, Puppet, Terraform) might not be enough to know

[1] https://www.entrepreneur.com/article/389265
[2] https://www.theverge.com/2019/3/14/18265185/facebook-instagram-whatsapp-outage-2019-return-back
[3] https://lists.openstack.org/pipermail/openstack-discuss/2021-January/019940.html
[4] https://stackoverflow.com/questions/34060140/what-config-file-does-openstack-poppy-use-how-to-log-the-debug-info-when-popp

all the configuration files for a project as certain configuration files might not follow a standard configuration format and can be as similar as a source code file (*e.g.*, python or bash script).

Existing studies on configuration maintenance are limited to a subset of the configuration files or types, and these studies' findings might not generalize to the whole configuration system. For example, the slow evolution of Dockerfiles and Docker-compose files observed by Cito et al. [12] and Ibrahim et al. [25] might not generalize, as Dockerfile and Docker-compose file are just two files among other files that contribute to a software system's configuration. For instance, a Dockerfile might be called from another configuration file (*e.g.*, Docker-compose, bash script, etc.), and a Dockerfile can invoke other files (*e.g.*, bash scripts, environment files), which are also used for the configuration of software infrastructure. Moreover, these files can be automatically generated from other files that might require more maintenance efforts. Hence, their findings of the estimation of configuration maintenance efforts are rather under-estimated, as there can be more files involved for the configuration of a software system.

Therefore, as a first step to improve the quality (*e.g.*, comprehension, maintainability, debugging errors) of the whole configuration system instead of single files or types of software configuration and to help developers better identify and understand the configuration system of a project, we first wish to understand what is a configuration system and how can we help developers and researchers automatically identify such a system. In fact, we aim in this paper to first understand what types of configuration files a software system (*e.g.*, OpenStack as our case study) can use during deployment and run-time, and how we can automatically identify configuration files and classify them under different types. To the best of our knowledge, no prior studies investigated what files can compose a configuration system, what purpose they serve, and how to identify and classify these files automatically. The main scope of our paper is on the configuration files that are used for deployment and run-time. Thus, we exclude configuration files associated with testing and build tools, such as Tox and Gradle.

Similarly to many previous studies [27, 45, 80, 89] that focused on studying OpenStack being the *"most widely deployed open source cloud software in the world*[5]*"*, we also focus on OpenStack. In this paper, we conduct an empirical study on the 635 projects of OpenStack to **(1) qualitatively identify what types of files can constitute a configuration system, (2) leverage machine learning models to automatically identify configuration files, and (3) classify configuration files into different types based on their usage in a software system life cycle**.

In particular, we define our first research question (RQ1) as follows:

- ***RQ1: What are the main types of files that constitute a configuration system?***
  Via a card sorting [91] qualitative analysis on 1,756 OpenStack configuration files, we identified nine configuration types that cover the deployment of OpenStack and its run-time customization. Our investigation reveals that these types of configuration files are interrelated. For instance, specific configuration files are being operated during the deployment and can create other configuration files that end-users can use during run-time. We also observe that besides infrastructure configuration files that are the direct inputs of the infrastructure as code (IaC) tools, other configuration files are also part of the infrastructure creation. **Our results suggest future studies to investigate the maintenance of the whole configuration system, rather than focusing on a subset of configuration files.**

Our qualitative analysis of RQ1 shows that identifying configuration files into different types is not straightforward. While we found that four types of configuration files could be directly identified from the OpenStack documentation, five types of configuration files can be written using the same programming language as ordinary source code files, such as Python and bash scripts. We also observe that the same file extensions are used to create different configuration types, which align with the hypothesis of the OpenStack developer, who stated

[5]https://www.openstack.org

that "*[he] do[es] not believe there is a fixed standard for configuration files vs. code*"[6]. For example, `Yaml` files can be used for three different types of deployment configurations. Therefore, we provide two machine learning models to identify which files are related to configuration, and then classify each identified configuration file into one of the nine types obtained from RQ1. We evaluate our machine learning models via the following two research questions:

- ***RQ2: How accurately can we classify configuration from non-configuration files?***
  Our model achieved a good performance in identifying configuration and non-configuration files, with a median AUC of 0.91, a median Brier score of 0.12, a precision that ranges between 0.78 and 0.86, and recall ranging between 0.77 and 0.83. We also observe that manually labeling a sample of data, as small as 100 files, would be enough to train a model that reaches a median AUC higher than 0.69. **Our results suggest future work to leverage our approach to identify configuration from non-configuration files**.
- ***RQ3: How accurately can we classify the different types of configuration files?***
  The combination of TF-IDF with the Random Forest (RF) into a multi-class classifier achieved a median weighted AUC of 0.92, a median Brier score of 0.04, a median weighted precision of 0.84, and a median weighted recall of 0.82. We also observe that one has to manually label only 100 configuration files to reach a median weighted AUC higher than 0.75. Similarly to RQ2, **we suggest future studies to leverage machine learning models to classify the configuration files into different types**.

Take-home message. While prior studies focused on the maintenance of software configuration, their evaluation cannot be generalized to the whole configuration system. Configuration systems involve several types of configuration files that are typically interrelated, for which we proposed two models to first identify, then classify into the nine types, respectively. Our first model requires labeling a minimum of 100 configuration and non-configuration files to a reach a median AUC of 0.69. Our second model requires labeling a minimum of 100 files to reach an AUC of 0.75. Our results suggest future studies on the maintenance of configuration systems **to first investigate what files constitute a configuration system for a case study and leverage a machine learning model to classify them**.

Replication package. Our replication package is publicly available for future replications [57]. In particular, we provide (*i*) our datasets, (*ii*) scripts for training machine learning algorithms, and (*iii*) details about the validation results along with the built models.

Paper organization. The rest of the paper is organized as follows. First, Section 2 presents a background about software configuration. Section 3 discusses our qualitative analysis and the nine types of configuration files that we identified. Section 4 discusses our automatic approach that leverages machine learning to classify a configuration file into the nine types. Section 5 discusses our threats to validity. Section 6 outlines existing studies on software configuration. Finally, we conclude the paper in Section 7.

2 BACKGROUND

This section aims to provide background about software and infrastructure configuration.

2.1 Background

We provide in this subsection further details about software configuration, as well as, the infrastructure-as-code tools (IaC) that are used for the configuration of infrastructures.

2.1.1 Software configuration. Software configuration is a mechanism used at different levels of a software life-cycle, from the customization of an infrastructure for the deployment to the customization of the execution of a

[6]https://lists.openstack.org/pipermail/openstack-discuss/2021-January/019940.html

software system at run-time [65]. For instance, a software system can specify its infrastructure requirements via a set of configuration files. For example, a developer can specify which software systems and libraries to install for the infrastructure of a software system. Similarly, developers can specify certain permissions and accesses, such as which users can access files and which ports can be exposed. All of these specifications can be written as code in dedicated configuration files. A developer can also specify the deployment of a software system in the Cloud, such as the number of instances for a software system and the amount of Cloud resources. Similarly, the end-users of a software system can customize its behavior at run-time using specific configuration options, which might be available under different formats. For example, a software system can provide a set of configuration options accessible to the end-users as a set of key and value pairs that are provided in different configuration files.

2.1.2 Configuration Infrastructure Tools. When developers write configuration files for the infrastructure; such files are used as input for tools dedicated to creating an environment. Several configuration tools exist for the creation of an environment, which are known as the Infrastructure-as-Code (IaC) [37] tools. Puppet [81], Chef [79], Ansible [23], and Terraform [7] are among the most popular IaC tools. Each of these tools can be used for different purposes of the infrastructure configuration. Some of these tools (*e.g.*, Terraform) are used to provision (*e.g.*, create a Cloud environment, set up a firewall) an infrastructure, which will be configured using other IaC tools (*e.g.*, Ansible) by installing a set of resources (*e.g.*, a web server) to deploy a software system. These tools heavily rely on configuration files that developers write for the deployment purposes of a software system.

While some existing tools expect configuration files with a predefined format, they represent only one type of configuration files and such files can still leverage other configuration files and can be invoked from other configuration files that do not necessarily follow a specific format or have a dedicated file extension. Thus, we aim by the following sections to investigate what would be these files that together constitute what we refer to as a "configuration system".

3 QUALITATIVE IDENTIFICATION OF THE CONFIGURATION SYSTEM

This section reports our qualitative analysis to understand the types of configuration files that constitute a configuration system. In particular, our qualitative study addresses the following research question:

RQ1: What are the main types of files that constitute a configuration system?

Motivation: This research question aims to qualitatively identify the different types of files that compose a configuration system. For instance, prior studies [12, 27] investigated the maintenance and evolution of different configuration files. However, they considered just a few types of files that typically use a specific format, making them trivial to identify, such as Puppet files (*i.e.*, configuration files with the ".pp" extension). Thus, prior studies' findings might not generalize to the whole configuration system. For example, while prior work by Hanakawa and Obana [22] observed that configuration files with the ".pp" extension rarely change, their findings are limited to only one type of configuration files and do not apply to the whole configuration system. Similarly, Jiang et al. [27] found the existence of some co-evolution trends between infrastructure configuration files and other artifacts (*e.g.*, build files). However, their findings of the co-evolution might be more important when studying the whole configuration system, which can mislead managers when estimating the cost of configuration maintenance. Meanwhile, there are no studies that identify all configuration artifacts that compose a configuration system. Therefore, this research question aims to qualitatively identify what types of files can compose a configuration system of a highly-configurable software system. While we expect to find some types of configuration files associated with a predefined file extensions, we aim at identifying whether there are other types of configuration files that exist and how difficult is it to identify them.

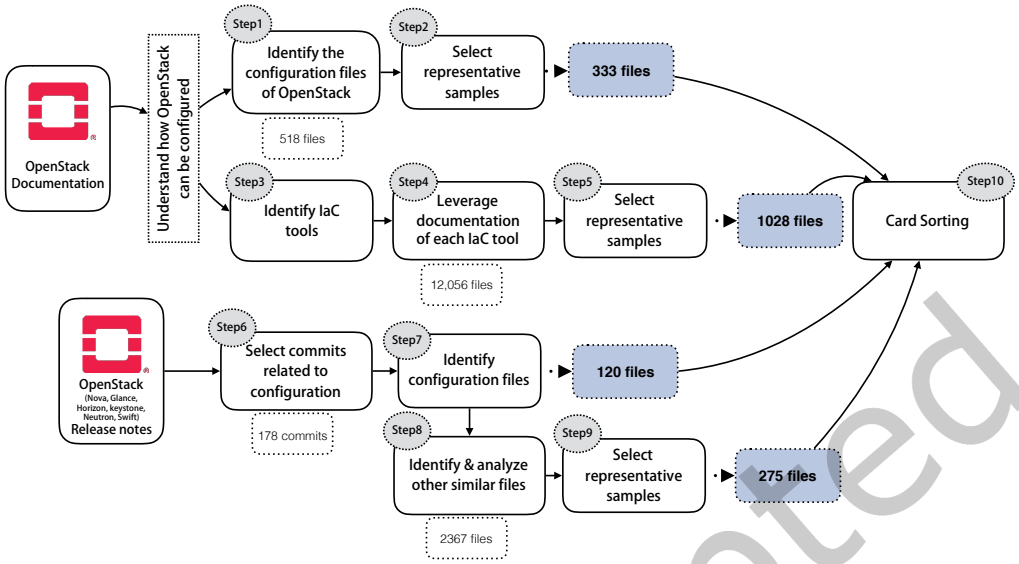


Fig. 1. Approach of the identification of configuration system (RQ1).

Approach: We qualitatively leverage the documentation and prior configuration-related changes (*i.e.*, release notes) to identify the various types of files that constitute the configuration system of OpenStack. From these two data sources (OpenStack documentation and release-notes), we obtain a set of files, mined from all OpenStack repositories. We aim to classify these obtained files under different types of configuration files using the Open Card Sort technique [91]. The Card Sort is a widely used qualitative method that consists of manually sorting similar data (*i.e.*, configuration files in our context) into clusters, which are then manually labeled according to the principal and common topic (*i.e.*, configuration file type in our context) to a cluster. Card sorting has become a standard tool and has been commonly used in a large body of qualitative studies in software engineering (*e.g.*, [39, 46, 64, 65, 71, 91]). The qualitatively studied configuration files were obtained by following different steps that are shown in Figure 1 and further explained as follows:

Leveraging documentation to understand the configuration of OpenStack: By manually investigating the documentation of OpenStack[7], we observe that it can be configured using a set of configuration options[8], which are defined in files that respect the `INI` file format (not only "`.ini`" files). We identified 518 files that respect the INI file format in OpenStack (step 1). These identified files result from the characteristics of files that are defined in OpenStack documentations. Then, we randomly select a statistically representative sample (step 2) out of the population size of the identified files (*i.e.*, 518 files) of Step 1. A confidence level of 95% and a confidence interval of 5% means that if we repeat the same experiment multiple times, we will reach the same conclusion with a margin of error of 5% in 95% of the cases. Eventually, we end up with a sample size of 333 files (step 2) to further investigate using the Card Sort technique (Step 10), as discussed at the end of this approach.

We also observed from the documentation that the infrastructure of OpenStack is typically configured and created by leveraging five different infrastructure-as-code (IaC) tools including Puppet, Chef, Ansible, Helm, and Juju (step 3). We then qualitatively investigate the documentation of each of these IaC tools to identify their

[7]https://docs.openstack.org/ocata/config-reference/
[8]https://docs.openstack.org/ocata/config-reference/config-format.html

Fig. 2. An example of a configuration related release note in the *Nova* project.

related configuration files (step 4). For example, the *Puppet* IaC tool is based on configuration files with the "`.pp`" extension. Similar to Step 1, we use a regular expression by which we identify 12,056 IaC potential files. From these files, we select a representative random sample (confidence level = 95%, and confidence interval = 5%) of 1,028 files (Step 5) to further investigate using the Card Sort technique (step 10).

Prior configuration-related changes: We extend our analysis by investigating the *release notes* of the six primary services of OpenStack (Glance, Horizon, Keystone, Nova, Neutron, and Swift)[9]. For instance, OpenStack attaches a release note file to the commits where a change related to "features, bug fixes, etc." is performed, which we leverage to identify configuration-related changes. Figure 2 presents an example of a configuration-related release note file[10] in the *Nova* project. In that release, developers add a new configuration option (*i.e.*, "`versioned_notifications_topics`") and update the appropriate files to access and use that new option in the source code. Similarly, they update the unit tests file *"test_rpc.py"* to test the new configuration option.

In particular, we select the releases with a release note that mentions at least one of the following configuration-related keywords: *(config*, infra*, setting, deploy*, setup)* (Step 6). We found 178 release notes (commits) with 350 co-modified files, which we manually examined to filter out files unrelated to configuration (Step 7). We ended up with 120 configuration files, which we added to our Card Sort analysis (Step 10).

We observed that bash scripts could also be used for configuration during our manual analysis of the configuration-related releases (Step 7). For example, in one of the studied release notes[11], we observe an interesting example of a bash script that is used for generating another configuration file. Therefore, we extended our analysis by studying bash scripts. Using file extensions, we identify 2,367 bash scripts (Step 8), from which we select a

[9]https://www.redhat.com/en/topics/openstack
[10]https://github.com/openstack/nova/blob/master/releasenotes/notes/Make-versioned-notifications-topics-configurable-a4baad995a74a076.yaml
[11]https://github.com/openstack/neutron/commits/master/releasenotes/notes/config-file-generation-2eafc6602d57178e.yaml

representative random sample (confidence level = 95%, confidence interval = 5%) of 331 bash scripts. We first manually analyze these bash script files to exclude 56 bash scripts that are not related to configuration and end up with 275 bash script files (Step 9) to qualitatively study using the Card Sort technique (Step 10).

Card sorting: Using a mixed card sort technique [91], we qualitatively classify the 1,756 configuration files into different types (Step 10). The mixed card sort combines both open and closed card sorting techniques, allowing us to create new categories while also utilizing a predefined set of categories. Precisely, we derive our categories based on the role of the configuration files. For each of the collected 1,756 configuration files from our manual analysis, we use the mixed card sorting technique to label the files under their correspondent types of configuration files using the following four steps. Note that we first understand the goal of a file by manually inspecting its content. From our comprehension of the types coming from the reading of the content of the files, from our inspection of the documentation and commit messages to select files (as discussed in our selection process), we also define under which phase each type of configuration file is used and what is the link between these types.

Step 10.1: Initial coding of configuration files. In the first iteration, each of the 1,756 configuration files that were obtained following the approach of Figure 1 are reviewed by two raters (*i.e.*, the first author and another co-author) independently. The first author manually labeled all the 1,756 configuration files, and each co-author labeled 585 files. In this iteration, we started with four labels (types of configuration files): externals, infrastructure-templates, infrastructure-variables, and infrastructure-setup. These types were identified from the documentation of OpenStack and the documentation of the IaC tools (*i.e.*, Puppet, Chef, Ansible, Helm, and Juju). In this first iteration, raters were allowed to add more types of configuration files. For each of the authors, it took approximately 15 work days to finish this step.

Step 10.2: Discussion of the labels (*i.e.*, types of configuration files). In this iteration, through several meetings, we discuss the types of configuration files that each rater identified, to end up with the nine types of configuration files.

Step 10.3: Review the types of configuration files. In this iteration, each rater reviewed their classification according to the finalized labels (*i.e.*, types of configuration files) of Step (b).

Step 10.4: Discussing disagreements. After Step (c), we measured the initial agreement/disagreement score between raters using the Kripperndorff's α [36]. When two authors tag the same file with two different labels, we consider it as disagreement. We achieved an initial agreement of α=0.79. To solve our disagreements, every two authors need to take a second round to discuss their understanding of the configuration goal of the file on which they disagree. If both authors still disagree on the classification of a configuration file, a third author is invited to the discussion. A final agreement is considered if at least two authors agree on the same category. Once all disagreements are re-evaluated, we re-calculate the Kripperndorff's α to check our final inter-rater agreement achieving an α = 0.95 which meets the reliability requirement ($\alpha \geq 0.8$) [36].

Results: Our card sort experiment revealed nine configuration files used for different configuration purposes and are interestingly interconnected. We define and provide an example for each configuration type in Table 1 and Table 2. We report and discuss our key findings of these nine types of configuration files in the following.

Finding 1: We observe that OpenStack has five types of files to configure it during deployment and five types to configure it during run-time, as shown in Figure 3. For instance, the *developers* of OpenStack prepare a set of configuration files that will be eventually used for the deployment of OpenStack. Such preparation requires defining the infrastructure on top of which OpenStack can be deployed. For example, such configuration files define the communication network between the services of OpenStack, decide which packages/dependencies to install or update, and specify the necessary resources (*e.g.*, memory) for a new version of OpenStack. A user, which we refer to as *administrator*, can decide to deploy OpenStack for their *end-users*. To do so, the administrator of OpenStack can then use the infrastructure-related configuration files to deploy OpenStack on a Cloud platform,

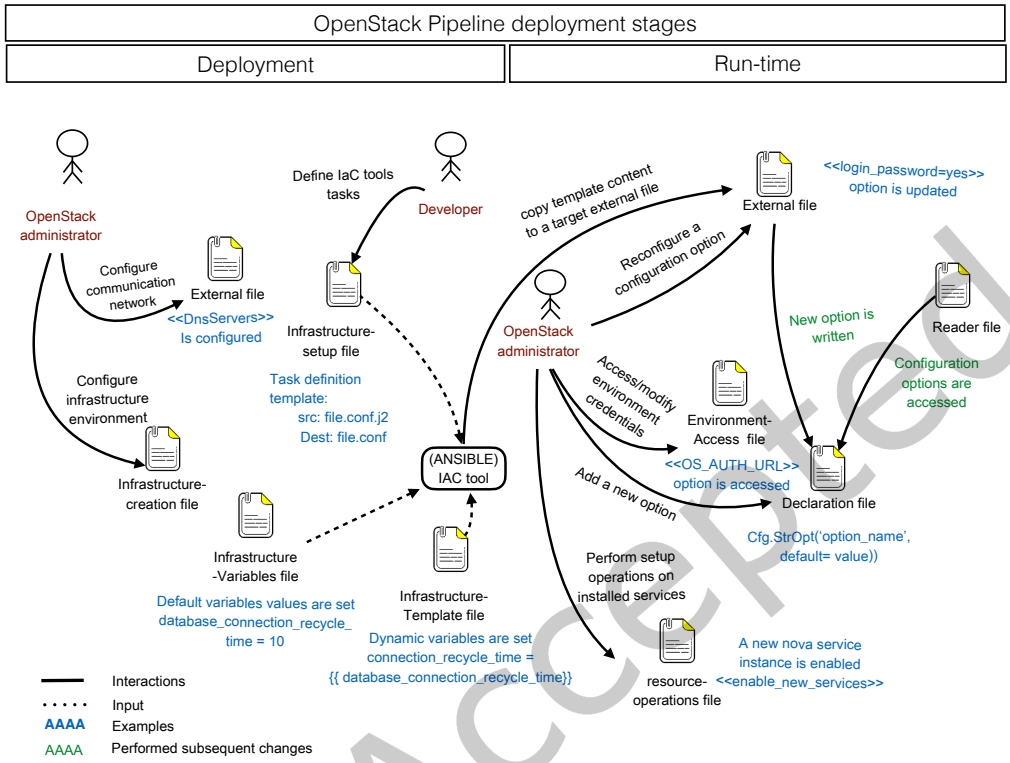


Fig. 3. A minimalist presentation of the files that compose the studied configuration system in OpenStack.

which the end-users will use. These deployment-related files are the input of a set of tools dedicated to creating the infrastructure (*i.e.*, IaC tools).

After deploying OpenStack, the administrators can still customize it using run-time configuration files. For example, one can configure the `ram_allocation_ratio` option to increase or decrease the number of instances running on the cloud. An administrator can also enable or disable a set of OpenStack services without redeploying a new version of OpenStack. We observe that both of these types (*i.e.*, deployment and run-time configurations) do not have just one type of configuration file, but multiple types of files, as discussed in the following.

Finding 2: We observe that five different types of files can be used to configure the infrastructure during deployment (*i.e., externals, infrastructure-creation, infrastructure-setup, infrastructure-templates,* and *infrastructure-variables).* Table 1 reports the different types of the identified configuration files, the definitions, and examples. First, OpenStack developers create a set of "*infrastructure-creation*" files to create the virtual environment (*e.g.*, python virtual environment) on the hosting servers. For example, the "build_venv.sh" file[12] is an infrastructure-creation file used to create a virtual environment. Interestingly, these files, including the last example, are not developed in a domain-specific language; instead, they are bash files.

[12]https://github.com/openstack/devstack/blob/master/tools/build_venv.sh

Once an environment is created, a set of scripts that we refer to as *"infrastructure-setup"* files are available to set up the infrastructure. These files are used for the customization of the infrastructure environment through the specification of different types of resources (*e.g.*, service, file, package, accounts, permissions, etc.) to be deployed. Other configuration files, which are used during run-time, can be used to control the behavior of the deployed resources, *e.g.*, by enabling/disabling services[13]. These infrastructure-setup files are written in a domain-specific language. For example, the *Puppet* IaC tool takes as input an infrastructure-setup file written in the Puppet programming language (a file with "pp" extension). During the deployment of OpenStack, users can define certain resource-related configuration options in a set of *"external"* files. For example, one can configure the memory required for the to-be deployed OpenStack release in an external file. Similarly, an external file can specify the network communication between the services of OpenStack. These configuration files could also be created by IaC tools to be used during run-time of OpenStack. Thus, the external files are shared between deployment and run-time of OpenStack.

During the execution of an IaC tool to create an infrastructure, the IaC tool can automatically generate new "*external*" configuration files from "*templates*" and "*variables*" files. Basically, the "*variables*" files are used to store dynamically generated values of settings based on the environment. These files will be later invoked by the "*template*" files that will describe the content of a file to be generated (*e.g.*, an external file), with the returned dynamic values. A template file can also be customized with a set of commands and if-conditions to customize the generated file to a specific environment. A typical example that can be coded in a template file is to attribute different IP addresses to different environments dynamically.

An example is shown in Figure 4, in which the infrastructure categories (infrastructure-setup, infrastructure-template, and infrastructure-variables) are combined to generate an external file. First, the "`keystone.conf.j2`" "infrastructure-template" file[14] defines the content of the to-be generated "*external*" file with a set of configuration options, such as "`connection_recycle_time`". These options are given dynamic variables (such as, "`{{database_connection_recycle_time}}`") in order to adapt the configuration options to different contexts. The dynamic value of the variable "`{{database_connection_recycle_time}}`" will be read from the "infrastructure-variables" file "*all.yml*"[15] during the creation of an environment using an IaC tool (*e.g.*, Ansible). Mapping a set of "infrastructure-template" files to the appropriate "infrastructure-variables" files can be customized from the "*infrastructure-setup*" files. In the "config.yml" "infrastructure-setup" file[16], users define a list of tasks to be applied on the hosting server. In our example, the tasks consist of copying over the "`keystone.conf`" file. Once this task is executed, the dynamic content of the "`keystone.conf.j2`" "infrastructure-template" file will be replaced by their values from the "all.yml" "infrastructure-variables" file. Finally, the IaC tool would copy the content of the "`keystone.conf.j2`" "infrastructure-template" file to the newly created "`keystone.conf`" "*external*" configuration file.

Finding 3: We observe five types (*i.e., external, reader, declaration, environment-access,* and *resource-operations*) of configuration files that are related to the configuration of OpenStack during run-time. We report in Table 2 the types of run-time configuration files. For instance, administrators can configure OpenStack through the configuration options defined in the "*external*" files, such as enabling the `login_password` option to enforce the authentication of the end-users of the deployed OpenStack. Furthermore, OpenStack administrators can configure a set of "*environment-access*" configuration files that define the required authentication information (*e.g.*, cloud URL, login, password) to access different clouds.

[13]https://puppet.com/docs/puppet/5.5/types/service.html
[14]https://github.com/openstack/kolla-ansible/blob/master/ansible/roles/keystone/templates/keystone.conf.j2
[15]https://github.com/openstack/kolla-ansible/blob/master/ansible/group_vars/all.yml
[16]https://github.com/openstack/kolla-ansible/blob/master/ansible/roles/keystone/tasks/config.yml

Table 1. Definition and example of the identified deployment configuration files.

Category	Definition	Example	Number

External	The external files customize a set of configuration options for OpenStack projects resources (*e.g.*, memory) that are required during deployment and run-time of OpenStack. An external file follows an INI file format with a `[group section]` describing the set of the `key = value` configuration options.	For instance, a user can specify how many times to reconnect with a server using the `http_request_max_retries` option in the external file `nova.conf` [56], which contains a set of configuration options as follows:
		`[keystone_authtoken]`
		`http_connect_timeout = None`
		`http_request_max_retries = 3`
			256

Infrastructure-Creation	The infrastructure creation are configuration files used to create the virtual environment where OpenStack will be deployed. These files are dedicated to bring up a complete virtual environment before the installation of OpenStack services using the infrastructure-setup files.	The `setup_pip.sh` [55] configuration file is dedicated to install the "pip" package management system for python, and specifies a set of configuration options that are related to pip. For example, the `setup_pip.sh` file contains the following snippet to upgrade the virtual environment:
		`# Upgrade to the latest version of virtualenv`
		`pip install --upgrade $PIP_ARGS virtualenv==20.7.2`
			125

Infrastructure-Setup	The infrastructure setup files are used to install different types of resources for OpenStack services on the environments created by the infrastructure-creation files. For example, the OpenStack Nova, Keystone, and Neutron services have their own dedicated infrastructure-setup files to install their required resources. Besides, these files are dynamic, as they could be used during the production environment to update resources state according to the requirements of the managed infrastructure.	The following example provides a minimalist code snippet of the IaC puppet "infrastructure-setup" configuration file *"init.pp"* [48], which specifies a file to be created under a certain directory with certain permissions for a nova service user.
		`file { '/var/lib/nova/.ssh':`
		`ensure => directory,`
		`mode => '0700',`
		`owner $=>$ $::nova::params::user }`
		The following example is a snippet from the IaC Ansible "infrastructure-setup" *config.yml* [51] file showing how an "infrastructure-setup" can involve an "infrastructure-templates" *keystone.conf.j2* file to get the *keystone.conf* external file (discussed in Table 2).
		`template:`
		`src: templates/keystone.conf.j2`
		`dest: keystone.conf`
			365

Infrastructure-Templates	IaC tools use "infrastructure-templates" to automatically generate external configuration files. These template files can have the static structure of an external configuration file with some dynamic values that can be stored in an "infrastructure-variables" configuration file. An "infrastructure-templates" configuration file can also implement a whole algorithm (*e.g.*, with if-checks and for loops) to automatically create an external file.	The *keystone.conf.j2* infrastructure template [52] defines a configuration option with a dynamic value. Thus, by running an IaC tool, the dynamic value will be replaced by a value that is stored in an "infrastructure-variables" file to automatically generate the *keystone.conf* external configuration file. `connection_recycle_time = {{ database_connection_recycle_time}}`	337

Infrastructure-Variables	IaC tools generate customized configurations to deal with the differences between software systems. This customized configuration is rendered in the "Infrastructure-Variables" files, where only configuration variables are defined. These variables will be dynamically assigned different values based on the system requirements. Finally, the "infrastructure-templates" will retrieve these dynamically assigned values to create the customized configuration file.	The dynamically defined variable in the *Infrastructure-Templates* file is stored in the *Infrastructure-Variables* file *all.yml* [53] with its default value, as follows:
		`#Database options`
		`database_connection_recycle_time: 10`
		`database_port: "3306"`
			326

Moreover, OpenStack centralizes the configuration options that administrators can change in a set of "*declaration*" files. The goal of these files is to define and load all the information about the configuration options coming from the command-line and parse them using a config manager. Such information consists of the name of the options, their types (*e.g.*, integer), whether they will be deprecated; if so, after which version, their detailed description, etc. These files prioritize different sources over each other. For example, if two values for the same option are provided from the command line and the configuration file, respectively, the value coming from the command line will be prioritized over the value coming from the configuration file. Thus, developers can add new configuration options to these files from the command-line, after checking whether there exists no similar configuration options to avoid duplicated or conflicting configuration options. Similarly, admins can use these "*declaration*" files to parse and understand the existing configuration options and how related they are. The declared configuration options in these files will be stored in a "CONF" object that loads the default values of all configuration options and will eventually be read from the "*reader*" files. These "*reader*" files are used to access all configuration options values from the "CONF" object defined by the "*declaration*" files.

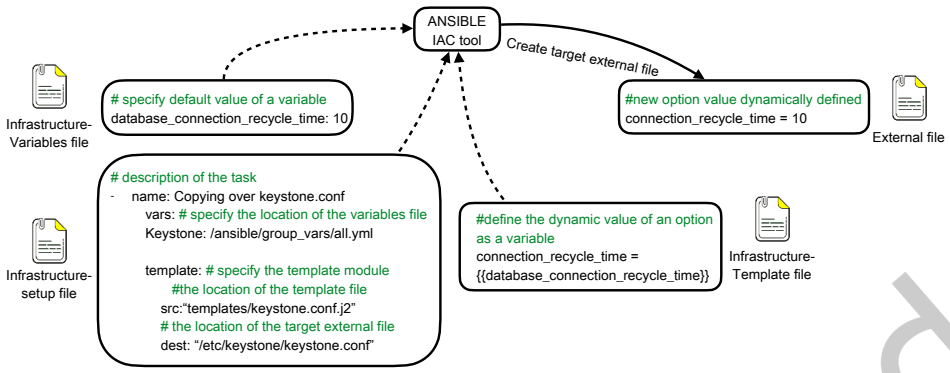


Fig. 4. A minimalist example of an "external" file creation through Ansible. An infrastructure tool (*e.g.*, Ansible) read the "*infrastructure-setup*" configuration file that specifies which "*infrastructure-templates*" configuration files to use with which "*infrastructure-variables*" configuration files.

Finally, one can perform resource-related operations (such as, enable/disable the service, create the service-related accounts and packages, upgrade resources, etc.) after deploying OpenStack. For instance, one can enable a new *nova-compute* service by setting the *"enable_new_services"* to true in the *"resource-operations"* files. Another example is the "plugin.sh" file [50], which contains a set of functions for operating the Aodh service. That file installs an Aodh client using the function "install_aodhclient()", create Aodh accounts using the "_aodh_create_accounts()".

Finding 4: Identifying the types of configuration files is not straightforward as several configuration files share similar formats with the source code files, and the same file format can be used for different types of configuration files. By manually investigating the extension of each of the nine types of configuration files, we searched whether each of the obtained extensions is used for any non-configuration files in the OpenStack project. Through this analysis, we observe that the nine types of configuration files have ten file extensions, as shown in Table 3. Seven file extensions can be used at the same time for a configuration file as well as a non-configuration file. For example, the "*reader*" and *"declaration"* configuration files are written in the same programming language (python) similar to any ordinary source code files. Similarly, we observe that three configuration files share the same file extension. For example, *"environment-access"*, *"infrastructure-setup"*, and *"infrastructure-variables"* can be all written in YAML files. However, we observe certain YAML files that are not for configuration. For example, the *"qemu-accept-vmxnet3-nic.yaml*[17] *"* is used for *"documentation"*. We also observe examples of files that are JSON files and not for configuration, such as the *"errors.json*[18] *"* file that is for logs of some test errors. The only type of configuration files that has two dedicated file extensions (".j2" and ".erb") is the *"infrastructure-templates"* which are templating engines. Thus, classifying different configuration files might not be as straightforward as looking at file extensions.

[17]https://github.com/openstack/nova/blob/master/releasenotes/notes/qemu-accept-vmxnet3-nic.yaml
[18]https://github.com/openstack/xstatic-angular/blob/master/xstatic/pkg/angular/data/errors.json

Table 2. Definition and example of the identified run-time configuration files.

Category	Definition	Example	Number
External	The external files define a set of configuration options to customize the execution (*e.g.*, timeout of shutting down a server instance) of OpenStack after its deployment.	A user can adjust the shutdown timeout of a server using the `shutdown_timeout` option in the external *nova.conf* file [56].	256

Environment-Access	The environment variables are the settings or rules that users need to run OpenStack or access OpenStack resources. For example, users must authenticate to the Keystone service to get authorization to perform an action. Thus, users invoke the environment variables to this end.	The following example is a snippet of the cloud configuration file *clouds.yaml* [54] containing all the settings required to connect to cloud instances.
		`clouds:`
		`  devstack:`
		`  auth:`
		`  auth_url:`
		`http://192.168.122.10:35357/`
			77

Reader	These are the source code that read the configuration options that are stored in the "CONF" object defined by the declaration files.	The following is an example of reading a configuration group `keystone_authtoken` defined in the reader configuration file *api_utils.py* [49] of the project *cinder*.
		`CONF = cfg.CONF`
		`CONF.import_group('keystone_authtoken')`
			56

Declaration	The declaration files are the source code files that host all the information about configuration options, such as their names, goals, comments, types, state, etc. as well as to which source code objects they are mapped (*i.e.*, in which option groups an option's value will be stored at runtime). To add a new configuration option, a developer declares it from the command-line and parse it from these declaration files using a config manager. Similarly, to understand configuration options, users can refer to these "declaration" files. These files can be considered as a "single-source-of-truth" for the configuration options of OpenStack.	In the following code snippet, from the *api.py* declaration configuration file [47], the `auth_strategy` configuration option, its default value, to which object it maps is declared:
		`auth_opts = [`
		`cfg.StrOpt("auth_strategy",`
		`default="keystone")]`
		`CONF.register_opts(auth_opt)`
			61

Resource-Operations	The goal of the resource-operations files is to perform operations after resource installation, such as enabling or disabling existing services, generating configuration files, upgrading resources, etc.	The *plugin.sh* file [50] has a source code snippet for disabling the *aodh* processes.
		`function stop_aodh {`
		`if [ "$AODH_DEPLOY" == "mod_wsgi"]; then`
		`disable_apache_site aodh`
		`fi }`
			153

Table 3. Extensions of the identified configuration and non-configuration files in our manually analyzed documents.

Types	.py	.j2	.conf	.sh	.ini	.yaml	.rb	.erb	.json	.pp	.aio
External			✓		✓						
Declaration	✓										
Reader	✓										
Resource_Operations				✓							
Environment_Access						✓			✓		
Infrastructure_Creation				✓							✓
Infrastructure_Setup						✓	✓			✓	
Infrastructure_Variables						✓					
Infrastructure_Templates		✓		✓	✓			✓			
Non-configuration files	✓			✓	✓	✓	✓		✓		

Summary for RQ1: Via a qualitative analysis of a representative corpus of 1,756 configuration files, we identified nine types of files that constitute the OpenStack configuration system. These nine types are scattered in different file extensions that are shared with the source code. This makes the distinction of different types of configuration files challenging, which motivates the need for an automated approach to classifying the configuration files. **Our results suggest that future research should consider the whole configuration system rather than a subset of configuration files.**

4 AUTOMATIC CLASSIFICATION OF OPENSTACK CONFIGURATION FILES

This section describes our approach to automatically identify (1) configuration from non-configuration files and (2) classify configuration files into the nine types of RQ1. The results of RQ1 revealed the co-existence

of nine different types of configuration files in a configuration system. Classifying these configuration files is important (1) for developers who need to continuously maintain and evolve their configuration systems, and (2) for researchers who want to investigate the whole configuration system rather than just certain configuration files. However, the classification of configuration files into different types is not straightforward, as some types of configuration files are often written in the same language as other ordinary source code files. In addition, different types of configuration files can use the same file extensions and formats (*e.g.*, python and bash files), as discussed in RQ1. The inherent challenges related to distinguishing configuration files from non-configuration files are also pointed out by OpenStack developers, as one of the developers stated that "*[he] do[es] not believe there is a fixed standard for configuration files vs. code*".

We leverage machine learning models to help practitioners and researchers identify different configuration files and classify them under the categories of RQ1. We opt for a machine learning model, instead of a simple grep or simply using the files extensions to identify configuration files and their types, as leveraging a machine learning model will be more usable, accurate, and requires less effort from practitioners compared to a keyword-based approach. For instance, we found in RQ1 that configuration files and ordinary source code files can share the same file's extension, such as ".py", ".sh", and ".rb", so using files extensions to identify configuration files or to classify them under different categories might not be accurate. On top of that, we observe some files with configuration related extensions (*e.g.*, ini), which are not for configuration. For example, the "tox.ini" file[19] is for testing and the "errors.json" file[20] is for logs. While typical configuration-related keywords like "conf*" could help find configuration files, some non-configuration files might include configuration-related keywords. Moreover, keywords could be susceptible to developers subjective judgements. That is, one needs to grep all possible variations of a keyword, which can lead to complex grep commands that can overlap for each category. On the other side, our goal of leveraging machine learning models it to automatically learn these keywords. We also think that classifying a configuration file under different categories by using keywords might be more challenging, as different types of configuration files can share keywords and such an approach will require developers to investigate different combinations of keywords. Therefore, we leverage machine learning models that learn these keywords and their weights that we investigate through the following research questions:

- RQ2: How accurately can we classify configuration from non-configuration files?
- RQ3: How accurately can we classify the different types of configuration files?

In the following, we address each research question. In particular, for each research question, we provide its motivation, our approach to answer the research question and the obtained results.

RQ2: How accurately can we classify configuration from non-configuration files?

Motivation: The goal of this research question is to help developers automatically identify configuration files since they have no standard format, are intermixed with ordinary source code files as found in RQ1, and no ownership exists over configuration files as suggested by Sayagh et al. [65] and for different scenarios which are discussed in our introduction. Automatically identifying configuration files can also help researchers who wish to study the maintenance of the whole configuration system instead of single files or types of configuration. Therefore, the goal of this research question is to investigate the performance of different machine learning models to classify configuration from non-configuration files. Moreover, we investigate the amount of efforts required to train a model with a decent performance. Specifically, we measure such efforts in terms of the number of files that one has to manually label to train a model.

Approach: In this research question, we leverage machine learning models to classify configuration from non-configuration files and measure the efforts required on manually labeling files for our classification models.

[19]https://github.com/openstack/cloudkitty/blob/master/tox.ini
[20]https://raw.githubusercontent.com/openstack/xstatic-angular/master/xstatic/pkg/angular/data/errors.json

Training and Testing our Classification Models: To automatically identify configuration from non-configuration files, we evaluate five machine learning models by following the approach depicted in Figure 5. Our dataset for training and testing our models consists of the 1,756 configuration files that we qualitatively studied in RQ1, as well as another set of 1,756 non-configuration files that we randomly selected from OpenStack (Step 1). Overall, our dataset consists of a total of 3,512 files. To construct our dataset for RQ2, we first needed to decide about the distribution of configuration files compared to non-configuration files to reflect real-life scenarios. Since the approach we followed to obtain the configuration files of RQ1 do not guarantee the identification of all the configuration files of OpenStack. Instead, the approach of RQ1 is just to cover enough files to manually study, for which we are sure that they are configuration files and represent a large enough distribution to identify most of the types of configuration files. Thus, we do not know the real distribution of configuration and non-configuration files in OpenStack. Therefore, we first opted for a balanced dataset distribution of 50% of configuration files and 50% of non-configuration files and also experimented with the ratio of 95%/5%, 90%/10%, 85%/15% down to 50%/50% of non-configuration/configuration files with a 5% decrement to check if our evaluation is consistent among different distributions of configuration and non-configuration ratios. As we did not observe any statistical differences, we decided to report our results based on the balanced dataset. We pre-process the 3,512 files (Step 2) to end up with a dataset, which we split into training and testing datasets (Step 3). Each file of our training dataset (*i.e.*, configuration and non-configuration) is transformed to a numerical vector by leveraging the TF-IDF algorithm (Step 4). We use these numerical vectors for training our models (Step 5-8), which we test by leveraging our testing dataset (Step 9). The following describes in details each of our nine steps for training and evaluating our machine learning models:

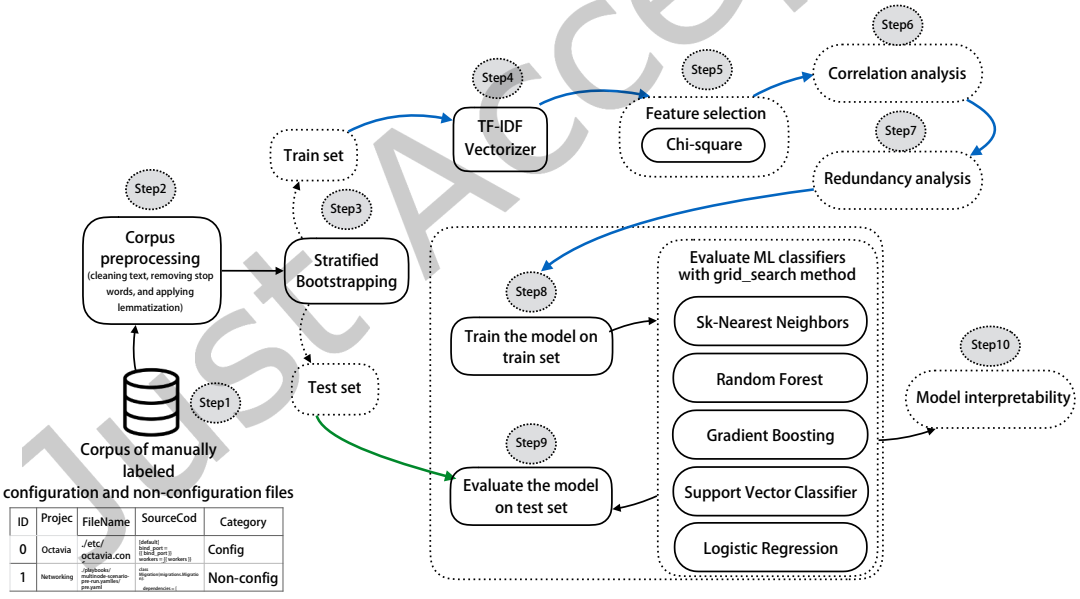


Fig. 5. Our approach to train and test our configuration and non-configuration files classifiers.

- *(Step 1) Files labeling:* We create a corpus of 3,512 configuration and non-configuration files to train our model. The corpus consists of the 1,756 manually inspected configuration files of RQ1. We label all of these files as "configuration", whereas we followed a semi-automated and manual iterative process to

identify 1,756 non-configuration files. To obtain these files, we first exclude all the files that were obtained during the process of our data collection of the configuration files, not just the 1,756 files that we manually analyzed (Step 1, 4, 6, and 8 of Figure 1). Such a process is to eliminate the maximum possible number of files that could be related to configuration. To further reduce the chances of selecting configuration files, we iteratively define a set of keywords (the final set of keywords that we used are: "conf*", "infra*", "parameter*", "setting*", "deploy*", "install*", and "setup") that should not be in the path of the selected non-configuration files. The iterative process consists of randomly selecting 100 files, manually analyzing each file by two authors to see if there is any chance for that file to be a configuration file. When doubting about a configuration file, we exclude it and update our set of keywords. We went through three iterations. These three iterations were made by the first two authors. We exclude any file on which the two authors either agree to be a configuration file, or at least one of the co-authors identify that file as configuration. As a result, we end up with a dataset divided into two groups (configuration and non-configuration), each with its textual content.

- *(Step 2) Text preprocessing:* similarly to many prior works [32, 63, 87] that modeled a textual content, we also preprocess the content of each file (*i.e.*, configuration and non-configuration files) through the cleaning of stop words (*e.g.*, I, and, does, have, etc.), which are not relevant to our two categories of files. We also consider the lemmatization of words, which converts each word to its basic form (*e.g.*, converting *enabled*, *enables* and *enabling* into *enabl*) such that TF-IDF does not consider the same word in different forms as different words.
- *(Step 3) Stratified Bootstrapping:* To make our evaluation statistically robust, we leverage 100 bootstrap samples for training and testing our models similarly to prior studies [29, 38, 73–75, 77, 78]. Basically, we select a bootstrap sample to train our model (following the Steps 4-8) and leverage the out of sample data to test that model (Step 9). We then repeat the same experiment 100 times, for each of which we leverage a different bootstrap sample.
- *(Step 4) Term Frequency-Inverse Data Frequency (TF-IDF):* We leverage TF-IDF to transform each file (configuration and non-configuration) into a numerical vector, which will be used as a data point for our model. We evaluated different word embedding techniques, such as Word2Vec and Doc2Vec. However, we kept the TF-IDF as it is simple, and it has been the most commonly adopted document representation technique and its performance outperformed other embedding techniques in similar studies [8, 17]. Besides, the TF-IDF suits our study as we are looking at giving a weight of importance to every feature based on its occurrence in a file category. For simplification, we report the results of just the TF-IDF. Basically, the dependent variable of our model is a boolean variable that indicates whether a file is a configuration or a non-configuration file. Our independent variables consist of the numerical vector that is generated by TF-IDF for each file. The TF-IDF vector for a file A consists of a set of numbers, each of which represent the relative relevance for a given word in A. Such a word's relevance is calculated based on the number of times a word appears in a file and how often the same word appears in the whole corpus of data (all of our configuration and non configuration files). The more important and unique a word is for a file, the higher the weight of that word for the same file and vice versa. For example, the word "`service`" is not unique to a specific configuration or non-configuration file, so its weight will be low, while the word "`opts`" is unique to a low number of configuration files, so its weight is supposed to be large. Since each vector generated by TF-IDF is influenced by the whole corpus of data on which TF-IDF is executed, we leverage TF-IDF just for the training set, so we train a model on data that has no information learned from our testing set.
- *(Step 5) Feature selection:* We reduce the number of features before training our models to avoid overfitting the model with many features over a few observations (*i.e.*, files). Thus, we improve our model training by focusing on relevant features for our observation (*i.e.*, configuration or non-configuration file). Previous research on text classification [11, 34, 58, 86] demonstrated that the *Pearson's chi-square* test [41] has a

positive impact on a model's performance as a feature selection technique. Pearson's chi-square would generate a contingency table of the frequency of features' occurrence in the classes (categories). We mainly focus on whether a feature (word) F_i and a category C_j are independent. If F_i and C_j are independent, we cannot use F_i to determine whether the document belongs to C_j. The Pearson's chi-square test measures the relevance between F_i and C_j. The higher the Pearson's chi-square score between F_i and C_j, the more relevant F_i is and the more probable to be considered as a relevant feature. Thus, all features of F_n will be sorted out based on their Pearson's chi-square scores to be selected as relevant for each category C_j. Note that we ensured that our data satisfies the assumptions of the Pearson's chi-square test [41]. In particular, no files belong to multiple categories (*i.e.*, no file is at the same time configuration and non-configuration file). In addition, in our experiments (100 bootstrap runs), the frequency of the selected features exceeded the value 5 in more than 95% of the contingency table cells, which satisfies the reliability requirement [41]. Thus, we use the *Pearson's chi-square* statistical test at a significance level of 95% to determine how correlated each independent variable is to our dependent variable. Note that we leverage the feature selection by leveraging only the training dataset. For example, the Pearson's chi-square test kept the configuration-related features: "`opt`", "`host`", "`default`", "`cfg`", and "`conf`" and discarded others such as "`language`", "`protocol`", "`api`", and "`order`".

- *(Step 6) Correlation analysis:* To deal with feature collinearity, we remove the correlated features as they can impact the interpretation of the models as pointed out by several prior works [28, 30, 42, 90]. We consider two features as correlated if they have a Spearman correlation coefficient higher than 0.7 [29, 30, 38, 72, 73]. Overall, we removed a median of 5 features throughout the 100 bootstrap samples. For example, our correlation analysis shows that each of the following pairs of features are correlated: ("`govern`", "`distribut`"), ("`self`", "`class`"), and ("`licens`", "`apach`").
- *(Step 7) Redundancy analysis:* Correlation analysis does not remove all the collinearity between the features. Thus, we remove independent features that can be redundant (being predictable by using other independent features). Thus, we further eliminate the redundant features as they can mislead the interpretation of our model [30, 42, 76]. The redundancy analysis iteratively trains different preliminary models, each of which predicts one independent feature using the other independent features. Then, it drops an independent feature f_i if it is predictable by the other independent features. In other words, if the preliminary model for f_i has a R^2 higher than 0.9, which is a similar threshold used by prior work [28, 30, 73]. The process stops once no feature can be predicted by other feature(s). We remove a median of one feature and a maximum of 3 (*e.g.*, "`base`", "`delet`", and "`note`") for our 100 bootstrap samples.
- *(Step 8) Machine learning classifiers:* From the previous steps, we obtained a set of final features that we use to train a machine learning classifier. In our approach, we leverage and evaluate five classifiers that are widely used by prior work [16, 29, 73] including Support Vector Classifier (SVC), Random Forest (RF), k-nearest neighbor (KNN), Gradient Boosting (GB), and Logistic Regression (LR). We compare between these different algorithms to select the most appropriate one for our classification problem. For each classifier, we also leverage the *grid searching* technique[21] to identify the optimal hyperparameters. The grid search calculates the performance for different combinations of hyperparameters values and returns the best values combination. In Table 4, we provide the parameters values for each classifier.
- *(Step 9) Models evaluation:* After training a classifier (Step 8), we measure its overall performance on classifying configuration from non-configuration files via the standard Precision and Recall measures. Furthermore, we use the performance measurements AUC and Brier Score, described below.

[21] https://scikit-learn.org/stable/modules/grid_search.html

Table 4. Classifiers parameters combinations returned by grid search technique.

Classifier	Parameters
Support Vector Classifier (SVC)	probability=True, C=1000, max_iter=-1, kernel='rbf', gamma='scale', decision_function_shape='ovr'
Random Forest (RF)	n_estimators=200, max_depth=20, random_state= 42
k-nearest neighbor (KNN)	n_neighbors=10, metric= 'euclidean', weights= 'distance'
Gradient Boosting (GB)	n_estimators=200, learning_rate=1.0, max_depth=10, random_state=42
Logistic Regression (LR)	random_state=42, C=50, penalty='l2', solver='newton-cg', max_iter=1000, multi_class= 'ovr'

  - Area Under the Curve (AUC): It represents the degree to which a model is capable of distinguishing between the different classes [5]. The higher the AUC, the better is the model. An AUC of 0.5 is similar to a random guess.
  - Brier score: The Brier score [6] assesses the difference between the true classes and their predicted probabilities (*i.e.*, the probability of each observation to be assigned to one of the two classes according to our trained model). The Brier score ranges between 0 and 1 and the lower the Brier Score the better the model is.

  By the end of the 100 bootstrap iterations, we obtain 100 AUC and 100 Brier Score performance measurements, which we leverage to statistically compare the five classifiers (discussed in Step 8) using the non-parametric Wilcoxon rank-sum test [14] and the Cliff delta non-parametric effect test [13]. The Wilcoxon rank-sum test is used to compare two independent samples and test whether these two samples derive from the same population by comparing their means. Then, we use the cliff delta effect size test to quantify the difference between two groups of observations. The effect size is considered negligible for d $<$0.147, small for d $<$0.33, medium for d $<$0.474, and large for d$\geq$0.47.
- *(Step 10) Model interpretability:* Similarly to previous studies [29, 31, 73, 85], we identify which features are the most important for classifying a file to better understand the behavior of our models. We use permutation analysis in particular as an in-built model technique that is well-used by prior work in software engineering [29, 31] to provide a global explanation of our models. The feature permutation consists of randomly shuffling the values of a feature and examining the impact of such a shuffling on the model's prediction. For each model, we end up with a ranking of the most important features. Since, we leverage 100 models (from the 100 bootstrap samples), we aggregate the 100 obtained rankings using the Scott-Knott clustering algorithm [26]. To examine which of the important features serve to predict each class, we create a new test data point with the median values of the features and compute its predicted probability p_1. Then, we modify the median value of one feature f_i, at a time, by adding one standard deviation and compute the new predicted probability p_2. Finally, we examine the difference between the two probabilities p_2 and p_1. If p_2 is superior to p_1, the feature is considered exhibiting a positive impact to predict the positive class (a file as configuration) and vice versa. We consider that the f_i feature has a positive contribution to predicting a class if the majority of our 100 models show a positive contribution. To better understand the performances of our model, we also investigate the files that are incorrectly classified. To do so, we select the files with the highest Brier score. These are the files with the largest differences between the actual class (configuration or non-configuration) and the probability score given by our models.

Efforts Analysis: We also evaluate how our models perform with different amounts of manually labeled configuration and non-configuration files, as a way to measure the manual efforts required to leverage our models. Previous studies suggested that the power of a machine learning model could be dependent on the size of the training sample [18, 35, 43, 44]. Thus, we evaluate what would be the minimal set of files to manually label to

obtain a model with a decent performance. To do so, we repeat the same previous steps with different data sizes, starting from 75 files. Note that these 75 files contain a set of configuration and non-configuration files. For each sample size (*e.g.*, 75), we executed 100 bootstrap iterations that select a sample of files and consider the previous steps (from Step 1 to Step 9). For example, for a sample size of 75, we select 100 different bootstrap samples of 75 files (configuration and non-configuration files) to train a model. We repeat the same experiment with other samples sizes.

Results: All of our evaluated models show a good performance and requires a reasonable effort of labeling files to reach a decent performance, as discussed in our two findings below.

Finding 1: Our evaluated models show a good performance with a median AUC ranging between 0.85 and 0.91, a median Brier Score that ranges between 0.16 and 0.12, a precision that ranges between 0.78 and 0.86, and a recall ranging between 0.77 and 0.83, as shown in Figure 6. The Random Forest (RF) model achieved the best performance in identifying configuration from non-configuration files, with a median AUC of 0.91, a median Brier score of 0.12, a precision of 0.86, and a recall of 0.83. The Gradient Boosting (GB) model delivered the second-best performance, with a median AUC of 0.90, a median Brier score of 0.15, and a precision and recall score equals to 0.83 and 0.82, respectively. The models LR, SVC and KNN show a similar performance with a median AUC between 0.85 and 0.86, a median Brier score of 0.16, and precision and recall scores between 0.77 and 0.79. We also observe a significant difference between RF and the three models (KNN, LR, and SVC) in the AUC and Brier score with (Wilcoxon test; p-values < 0.05) with large associated effect sizes. However, the RF model shows a negligible difference in AUC score with the GB model and a large difference in the Brier score. Even though the models performed differently in the AUC and Brier scores, they all leveraged a good configuration and non-configuration files classification. We further report that the RF model is better in terms of training and testing time, with less than 40 minutes (on a laptop). In contrast, the GB model that achieved the second-best performance requires at least 1 hour. As the RF model achieved the best performance, we evaluate all OpenStack projects source code files to determine the distribution of configuration files. The outcome of this analysis revealed that the configuration files account for 17.26%.

Our permutation importance analysis reveals that the semantics of five out of the top-10 most important features are related to configuration and contribute to the classification of files as configuration files. The remaining five important features are not semantically related to configuration and contribute to the classification of files as non-configuration files. In fact, among these ten most important features, we do not observe any configuration-related word that is important to classify a file as a non-configuration file and vice versa. As shown in Figure 7, the configuration related words (highlighted in red) that contribute to classifying files as configuration are: "`configur`", "`conf`", "`host`", "`install`", and "`enabl`". The keyword "`configur`" (ranked second) can be present in any configuration script to denote a configuration function or even code comments to describe a configuration task. The "`conf`" (ranked 3rd) is an object from the `Oslo_config`[22] framework to access a configuration file. Besides, the feature "host" (ranked seventh) represents an essential keyword in the infrastructure-setup configuration files to define the host group name. The host typically defines the environments (*e.g.*, web or database servers) on which Ansible tasks will take effect. Furthermore, the words "install" (ranked seventh) and "enabl" (ranked ninth) could denote any resource-related configurations. On the other hand, we observe that the non-configuration related words (highlighted in green), "`arg`", "`check`", "`instanc`", "`copi`" and "`auth`" serve to predict the non-configuration class.

Seven of the top-10 misclassified files are not configuration files, but contain configuration-related words. For example, the non-configuration file *requirements.txt*[23] contains words related to configuration such as "`config`", so our model incorrectly classifies that file as a configuration while it is a text file. The same applies to the file

[22]https://docs.openstack.org/oslo.config/queens/reference/cfg.html
[23]https://github.com/openstack/oslo.tools/blob/master/requirements.txt

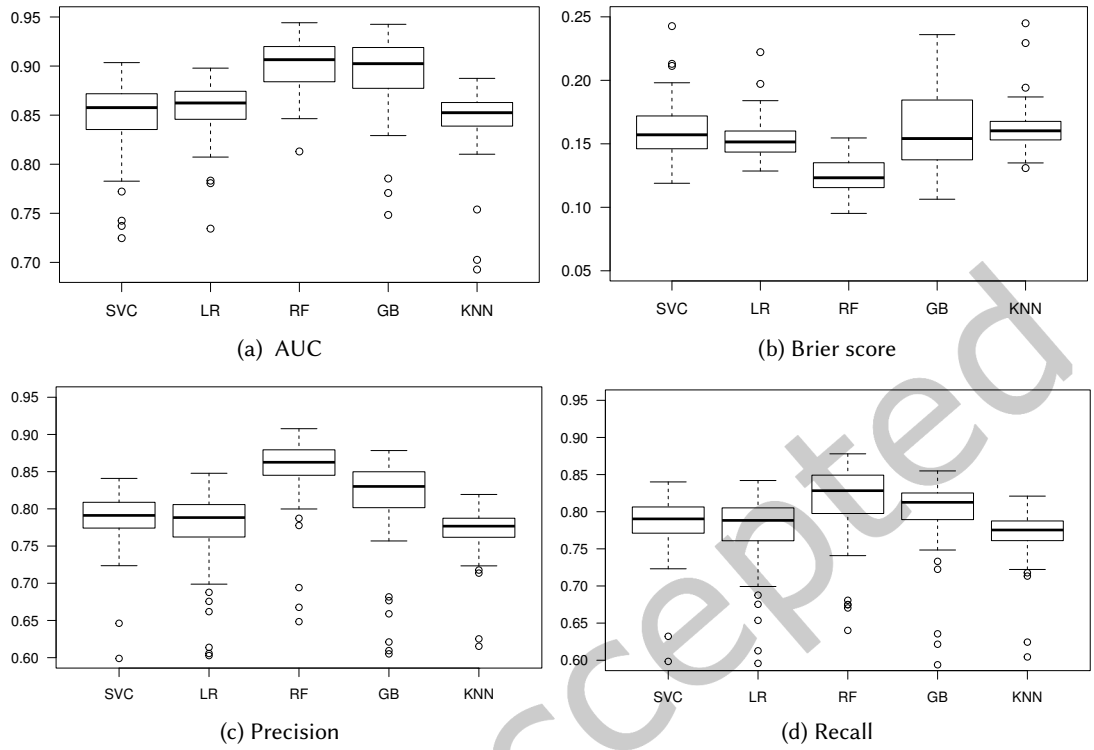


Fig. 6. he obtained model's performance results in classifying configuration and non-configuration files in terms of AUC, Brier Score, Precision, and Recall.

rbd.py[24] where it incorporates configuration-related keywords, such as "`conf`" and "`default`". Furthermore, we notice that our model mistakenly classifies the release-notes[25],[26] and "rst" files[27] as configuration files. These files contain words such as "`host`", "`parameter`", "`option`" and "`enable`" that denote a configuration code. As another example, the configuration file *systemd-resolved.conf.j2*[28] do not include any configuration-related keyword, so our model incorrectly classified it as non-configuration files. Furthermore, the files *main.yml*[29] and *os-faults.spec.j2*[30] are misclassified as they contain non-configuration related keywords such as, "`check`" and "`import`".

Finding 2: We observe that we can reach a median AUC of 0.69 by labeling just 100 configuration and non-configuration OpenStack files. We present in Figure 8 the five models performances with respect to different sample sizes in terms of AUC, Brier score, Precision, and Recall. We observe in Figures 8a, 8b, 8c, and 8d

[24]https://github.com/openstack/os-brick/blob/master/os_brick/privileged/rbd.py
[25]https://github.com/openstack/puppet-nova/blob/master/releasenotes/notes/nova_scheduler_limit_tenants_to_placement_aggregate-8886c514f0ebbb72.yaml
[26]https://github.com/openstack/nova/blob/master/releasenotes/notes/microversion-2.43-77d63cae38695fd1.yaml
[27]https://github.com/openstack/security-doc/blob/master/security-guide/source/compute.rst
[28]https://github.com/openstack/ansible-role-systemd_networkd/blob/master/templates/systemd-resolved.conf.j2
[29]https://github.com/openstack/ansible-hardening/blob/master/tasks/rhel7stig/main.yml
[30]https://github.com/openstack/rpm-packaging/blob/master/openstack/os-faults/os-faults.spec.j2

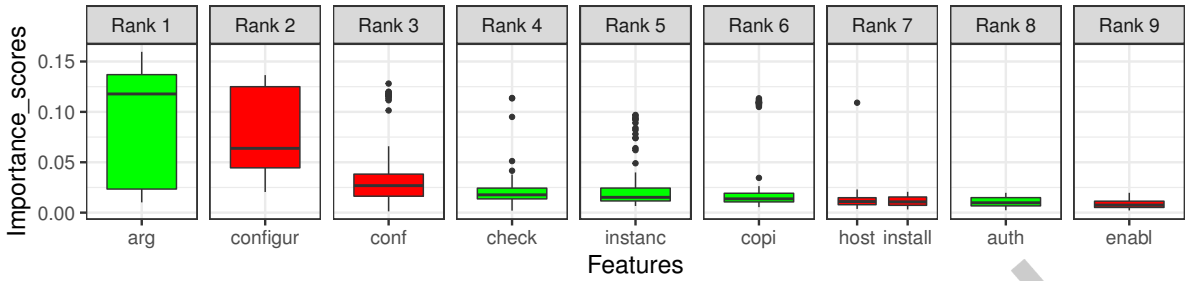
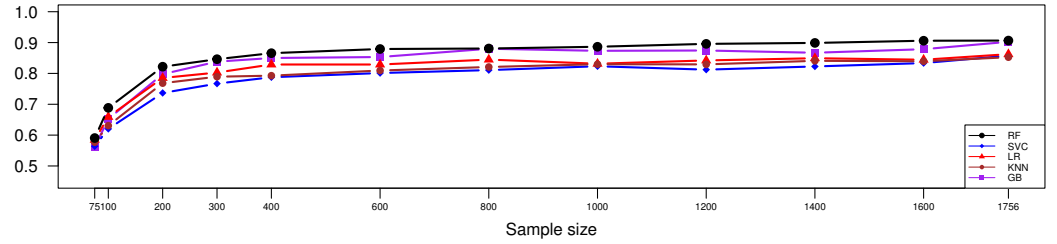
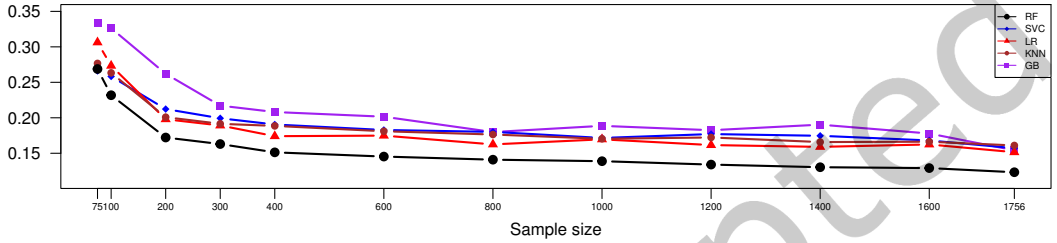


Fig. 7. Scott-knott ranking of the top-10 most important features of configuration and non-configuration files identified in 100 bootstraps. The red color indicates features that contribute to the classification of a file as a configuration file, while the green color is used for features that contribute to the classification of files as non-configuration files. For example, the "charm" feature is the most important feature, and it contributes to the prediction of files as configuration files. The higher the value of the "charm" feature, the higher the probability of a file to be classified as a configuration file. On the other side, the higher the value of the "def" feature, the lower the probability of a file to be classified as a configuration file.

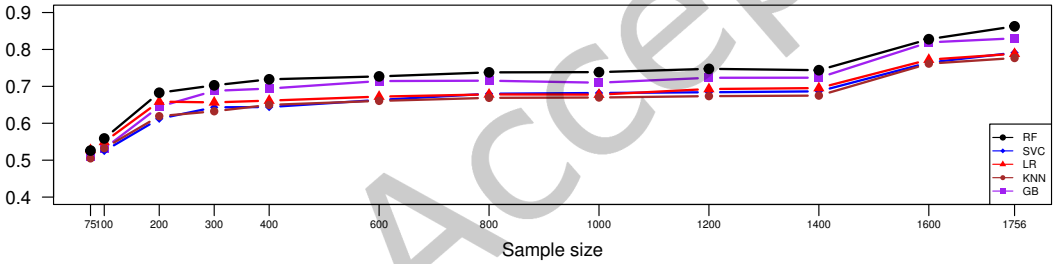
that the RF model requires minimal effort than the other models to achieve a decent classification. We only need to manually label 100 files to reach a median AUC higher than 0.69, a median Brier score less than 0.23, a median precision and recall of 0.56 and 0.55, respectively, with the RF model. The KNN and LR models need at least 100 files to reach a median AUC higher than 0.63 with a median Brier score less than 0.26, and a median precision and recall higher than 0.53. The SVC model needs 100 files to reach a median AUC higher than 0.63 and a median Brier score of 0.26, and a median precision and recall higher than 0.52. We also notice that the GB model needs a set of 100 manually labeled files to reach a median AUC higher than 0.65 with a median Brier score of 0.37, with a median precision and recall higher than 0.53. Overall, the RF model requires less manual labeling effort of 100 files to achieve a decent classification with a median AUC higher than 0.69. We conclude that the RF model is not just the fastest model to train/test according to finding 1, but it is also the model that requires less effort to achieve a decent classification performance of configuration and non-configuration files according to finding 2.



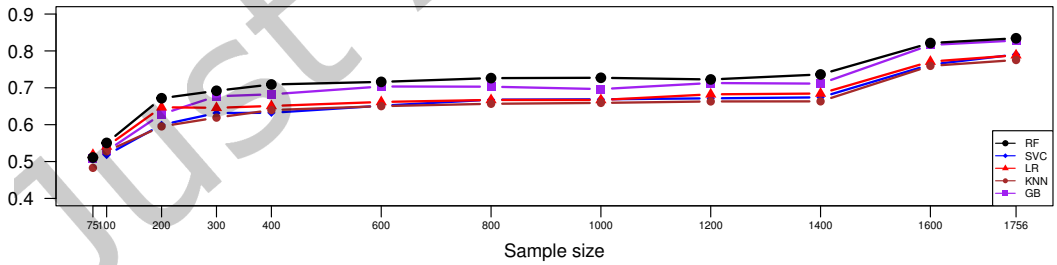
(a) AUC



(b) Brier score



(c) Precision



(d) Recall

Fig. 8. Comparison of sample sizes impact on the performance of the five classifiers (RF, SVC, LR, KNN, and GB) in terms of AUC, Brier score, Precision, and Recall.

Summary for RQ2: Our RF classification model that is trained on TF-IDF dataset shows a good classification performance of the OpenStack configuration and non-configuration files, with a median AUC of 0.91, a median Brier score of 0.12, and median precision and recall of 0.86 and 0.83, respectively. Our RF model can reach a median AUC of 0.69 using just 100 manually labeled OpenStack files.

RQ3: How accurately can we classify the different types of configuration files?

Motivation: While a previous study [21] found that one of the most common IaC issues is that dealing with the different formats of IaC tools is often obscure to most and requires specialized personnel, we believe that the problem can be more challenging when considering the whole configuration system, not just IaC, of a software application. Therefore, to help developers identify different types of configuration files and toward a first step for further studying different formats of configurations (not just IaC configurations), we develop a machine learning model that identifies these files automatically to help developers identify them from a plethora of configuration files that a system can have. Identifying different types of configuration files can also assist researchers in conducting more focused studies on specific file types that may be more challenging to identify, understand, or maintain than those previously studied. This research question aims to evaluate how different machine learning techniques leverage the textual content of a configuration file to classify it based on the nine types identified in RQ1. Similarly to RQ1, we also evaluate the trade-off between performance of our classification models and the amount of efforts required on classifying configuration files to train a model.

Approach: We formulate our problem as a multi-class classification task to automatically categorize the various configuration file formats. The purpose is to categorize a given file into a specific configuration type based on its content. Each file is classified into exactly one type. Classifying files is considered supervised learning, as we will use machine learning techniques to classify files based on the labeled configuration files in RQ1. We apply the same classification approach adopted in RQ2, with the following two adjustments regarding data balancing and support of multi-class classification.

- *Data balancing:* To address the uneven distribution of our file types in the corpus, we employ the SMOTE oversampling approach [9]. SMOTE will balance the distribution of classes by producing new instances for minority types. We only balance our train set in this step.
- *Support of multi-class classification:* Similarly to RQ2, we compare the performance of the various classifiers including RF, KNN, GB, LR, and SVC. However, as LR and SVC do not natively support multi-classification problems, we employ the *One-vs-Rest (ovr)* ensemble technique, which entails dividing a multi-class dataset into multiple binary classification problems [20, 61]. We then evaluate their performance using the weighted Precision, weighted Recall, AUC, and Brier score measurements. We further report the *Weighted Area Under the Curve* (weighted AUC) [40], which weights the score by computing the number of valid instances for each class.

To get a more qualitative sense of the built models, we further investigate the ranking of the most important features the model relies on to classify the different types of configuration files. We then investigate the possible reasons behind the model's misclassified instances by manually examining ten files with the highest Brier scores.

We finally assess the minimal required effort, in terms of the number of labelled configuration files, to achieve a decent classification of each model. In this case, we assume that developers/researchers already identified what files are for configuration and wish to classify these configuration files under one of the nine different categories. To do so, We select different samples of configuration files starting from 75 configuration files and measure our models' performances. To make our analysis statistically sound, we select 100 different samples of the same sample size (*e.g.*, 75) of configuration files and evaluate the performance of our models.

Results: The five classifiers achieved a good performance in identifying the different configuration types in terms of weighted AUC, standard AUC, and Brier score as shown in Figure 9. Similarly to RQ1, our models do not require manually labeling many files.

Finding 1: The multi-class classification models achieve a good classification performance with a median weighted AUC ranging from 0.86 to 0.92, a median Brier Score ranging from 0.04 to 0.09, a median weighted precision ranging between 0.76 and 0.84, and a median weighted recall ranging between 0.74 and 0.82 for all the configuration types. Mainly, we observe that the Random Forest (RF) model delivers the best performance with a median weighted AUC of 0.92 and a median, a median Brier score of 0.04, a median weighted precision of 0.84, and a median weighted recall of 0.82, followed by the SVC model with a median weighted AUC of 0.88, a median Brier score of 0.06, a median weighted precision of 0.83, and a median weighted recall of 0.81. Furthermore, we observe similar good results with the GB and LR models achieving a median weighted AUC of 0.87, a median Brier score of 0.07 and 0.08, respectively, and a median weighted precision of 0.77 and 0.76 with a median weighted recall of 0.74 and 0.74, respectively. The KNN model achieves a median weighted AUC of 0.86, a median Brier score of 0.07, a median weighted precision of 0.76, and a median weighted recall of 0.74. Furthermore, we find a statistically significant difference (Wilcoxon test; p-value < 0.05) between the RF and the rest of the models in the standard AUC, weighted AUC, and Brier scores with a small to medium effect sizes. We also observe that the RF model is the best in the training and testing time, taking approximately 35 minutes (on a laptop). Therefore, we advocate to experiment with the RF algorithm on such a multi-class classification problem.

Given that the RF model outperformed the other classifiers, we use it to identify the types of all configuration files within OpenStack projects that were identified by our binary model. As a result, we found the following distributions of the nine configuration types, as shown in Figure 10. As we can clearly observe, the infrastructure-Setup, infrastructure-Variables, and infrastructure-Templates configuration types are the most prevalent, accounting for over 80% of the entire configuration system.

We find that the top-10 features serve to predict six configuration file types *"external, declaration, infrastructure-setup, environment-access, resource_operations"*, and *"reader"*. We observe from Figure 11 that the first ranked feature "`default`" (highlighted in orange) serves to predict the *external* configuration file; *i.e.*, the higher the value of "`default`", the higher the probability of a configuration file to be for the external type. The keyword "`default`" denotes the group name of the default configuration options in the external configuration files, following the syntax "`[Default]`". Besides, we find that the keywords "`opt`" and "`list`" (highlighted in green) predict the *declaration* configuration file. For example, the feature "`opt`" (ranked 8th) is a keyword representing the class used to define the configuration options[31]. Furthermore, we find three features serving to predict the *infrastructure-setup* file, including "`ansibl`", "`host`" and "`kolla`" (highlighted in blue). For instance, the keyword "`ansible`" could be defined in the Ansible IaC tool playbook name[32] of the Ansible IaC tool in the infrastructure-setup. The feature "`host`" (ranked 9th) represents an essential keyword in the infrastructure-setup configuration files to define the host group name. The host typically defines the environments (*e.g.*, web or database servers) on which Ansible tasks will take effect. The features "`openstack`" and "`type`" (highlighted in pink) contribute to the prediction of the *environment-access* type, whereas "`package`" (highlighted in yellow) serves to predict the *resource_operations* type. Finally, the feature "`conf`" (highlighted in red) contributes to the prediction of the *reader* configuration type. The feature "`conf`" (ranked fourth) is an object from the `Oslo_config` module to access the *external* files from a *reader* configuration file. These features only represent the top-10 features with the highest impact on the model's performance,*i.e.*, other relevant features could contribute to the prediction of

[31]https://docs.openstack.org/oslo.config/queens/reference/opts.html#oslo_config.cfg.Opt
[32]Playbooks are the Ansible IaC tasks responsible for executing Ansible's configuration and deployment functions: https://docs.ansible.com/ansible/latest/user_guide/playbooks.html

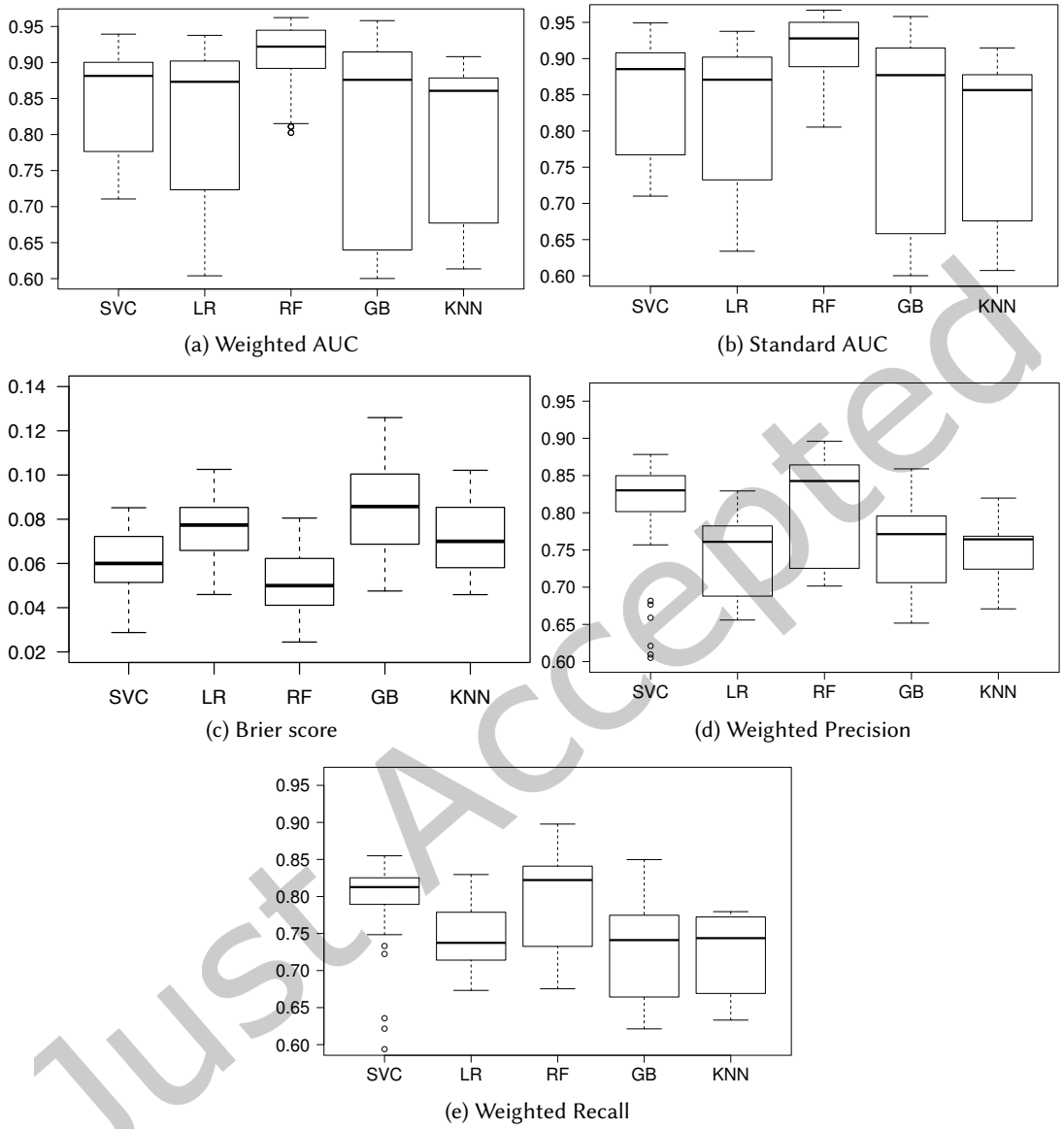


Fig. 9. The five classifiers performance in predicting configuration file types in terms of weighted AUC, standard AUC, Brier score, weighted Precision, and weighted Recall.

the remaining configuration types: *infrastructure_variables, infrastructure-template* and *infrastructure-creation*. Overall, our analysis reveals that the ten most important features to predict the types of configuration files are indeed words that are relevant to their respective type of configuration files.

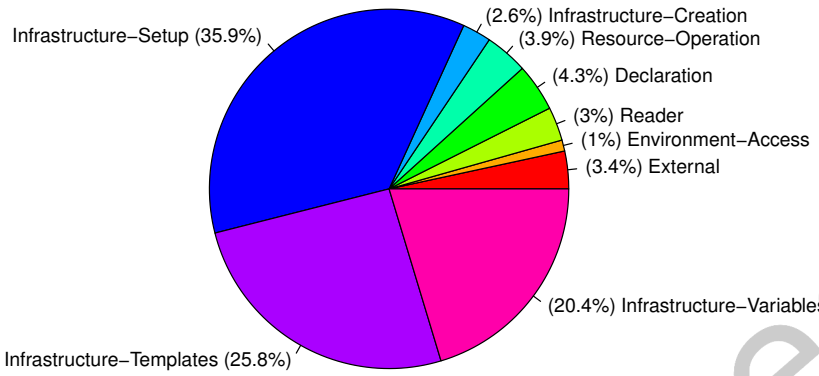


Fig. 10. The distribution of the identified nine configuration file types in OpenStack projects.

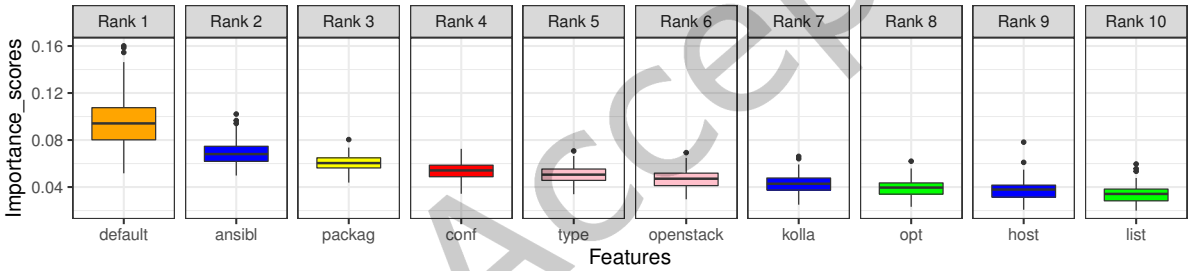


Fig. 11. Scott-knott ranking of the top-10 most important features of configuration file types identified in 100 bootstraps. The orange color is used for features contributing to the prediction of the external configuration type. The green is for features that serve to predict the configuration declaration type. The blue color represents the features that contribute to the prediction of the infrastructure-setup type. The pink color is for features contributing to the environment_access configuration type. The yellow color represents the features predicting the resource_operations configuration files, while the red is for features contributing to the prediction of the reader configuration files.

We find in the top-10 misclassified files three *infrastructure-setup* files, two *declaration* files, two *external* files, one *infrastructure-templates* file, one *infrastructure-variables* file, and one *resource-operations* file. Taking the examples of the infrastructure-setup file *enroll-dynamic.yaml*[33], the infrastructure-templates *pycadf.spec.j2*[34], the infrastructure-variables *ssh.yaml*[35], and the external *glance-scrubber.conf*[36] file, we observe that they do not incorporate any feature related to their type, which could lead the model to misclassify the files. Furthermore, some configuration file types could include keywords that frequently appear in another configuration file type. For

[33]https://github.com/openstack/bifrost/blob/master/playbooks/enroll-dynamic.yaml
[34]https://github.com/openstack/rpm-packaging/blob/master/openstack/pycadf/pycadf.spec.j2
[35]https://github.com/openstack/charm-openstack-dashboard/blob/master/charmhelpers/contrib/hardening/defaults/ssh.yaml
[36]https://github.com/openstack/glance/blob/master/etc/oslo-config-generator/glance-scrubber.conf

instance, the template file *pycadf.spec.j2*[37] includes the feature "`group`", which is usually found in the declaration configuration type.

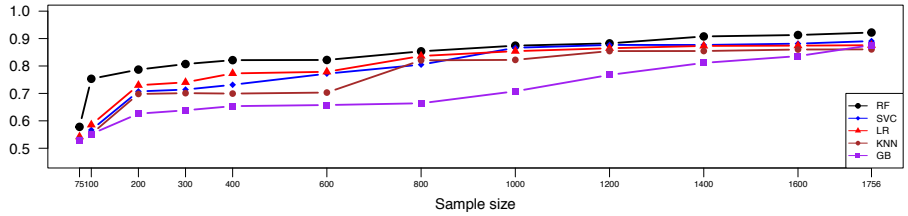
Finding 2: Our model can reach a median weighted AUC greater than 0.75 with only 100 OpenStack configuration files. Figure 12 shows a comparison between the five classifiers in terms of the weighted AUC, standard AUC, Brier score, weighted Precision, and weighted Recall for different sample sizes. We observe in Figures 12a and 12b that each classifier relatively shows similar performance results in the weighted and standard AUC metrics. We observe that the RF model needs less effort of manually labeling files to achieve a decent performance compared to the other models. Specifically, the RF model only needs 100 files to achieve a median weighted AUC greater than 0.75 with a median Brier score of less than 0.1, a weighted precision of 0.57, and a weighted recall of 0.54. The SVC model needs 200 files to achieve a weighted AUC higher than 0.70 with a median Brier score less than 0.1, and a median of weighted precision and weighted recall more than 0.55 and 0.53, respectively. We also observe that the LR, KNN models need at least 200 files to achieve a median weighted AUC higher than 0.70, a median Brier score less than 0.1, and a weighted precision higher than 0.56 and a weighted recall higher than 0.53, respectively. The GB model needs at least 1000 files for an AUC higher than 0.70. Overall, the RF model can deliver decent classification performance of the nine configuration file types with minimal effort of only 100 manually labeled files. Finally, we report that the RF model is the fastest in training and testing time, in addition to that, it requires the least number of manually labeled files to achieve a decent classification performance of the different configuration file types.

While we admit that one might not end up selecting all the nine types in their random sample of configuration files, the chance of missing at least one type is 0.35, 0.32, 0.28 0.27, 0.21, 0.23, 0.19, 0.16, 0.17, 0.13, 0.09, and 0.07 for a sample size of 75, 100, 200, 300, 400, 600, 800, 1000, 1200, 1400, 1600, and 1756 configuration files respectively. To identify these last probabilities, we selected 100 different samples of 75 (as an example) configuration files and identified how many configuration types were obtained in that sample. We observe that 65 out of the 100 samples have all the nine types of configuration files, 6 out of the 100 are missing one file. That said, 71% of the samples have at least 7 types of configuration files. Similarly, the larger the sample size, the more likely a selected sample will cover all the types of configuration files.

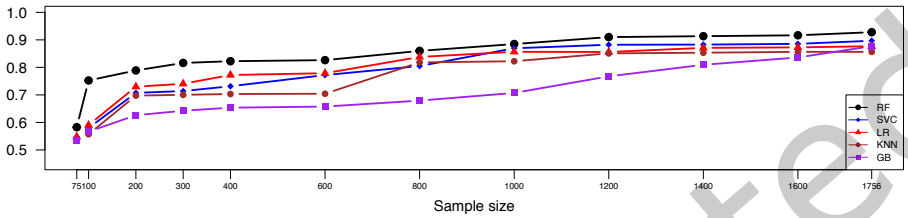
Summary for RQ3: Our RF model can classify an OpenStack configuration file into one of the nine types of configuration files with a median weighted AUC of 0.92, a median Brier score of 0.04, a median weighted precision of 0.84, and a median weighted recall of 0.82 using the RF algorithm. Our model can also reach a median weighted AUC higher than 0.75 with 100 manually labeled OpenStack' configuration files.

Discussion: By looking at the documentation of OpenStack, we did not find 5 types of configuration files. Among these types, some files are not easily distinguishable from ordinary source code files, such as python and bash files. Therefore, our models can be used by researchers and developers for better maintenance of configuration files. (1) Identifying files that are for configuration will help propose better approaches to maintain them. For instance, researchers can use our model to identify all the configuration files instead of single files (Dockerfile, IaC files) so they can generalize their results to the whole configuration of a system rather than a few files. For example, Jiang and Adams [27] found that infrastructure as code files co-evolve with code and test files to help managers better estimate the cost of maintaining configuration files. However, such a cost can be significantly under-estimated, as that prior work focused just on a subset of the whole configuration files. Thus, future studies can better help developers to better maintain their systems. (2) According to a discussion with an OpenStack

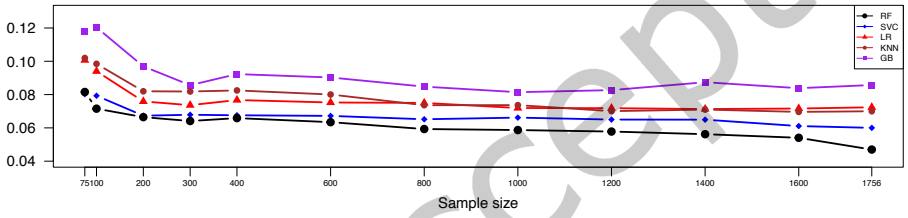
[37] https://github.com/openstack/rpm-packaging/blob/master/openstack/pycadf/pycadf.spec.j2



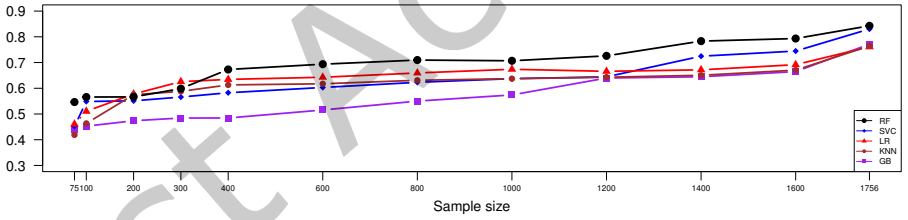
(a) Weighted AUC



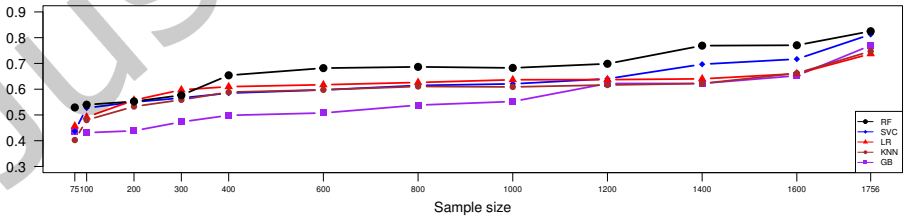
(b) Standard AUC



(c) Brier score



(d) Weighted Precision



(e) Weighted Recall

Fig. 12. Comparison of sample sizes impact on the performance of the five classifiers (RF, SVC, LR, KNN, and GB) in terms of weighted AUC, AUC, Brier score, weighted Precision, and weighted Recall.

developer, we observed that no standards exist for configuration files, so developers can leverage our model to identify which configuration files exist. We also observe that developers consider just files that have the ".conf" extension as configuration files, while our study shows the existence of other files for configuration with different file extensions, including code extensions. In fact, there are files that are written in python and bash scripts that can be for different types of configurations, for which we built two machine learning classifiers to identify them. Our model can be also used by novices to quickly distinguish between configuration and non-configuration files and understand a software system.

We believe that a model with an acceptable performance is better than randomly guessing which files are for configuration, which can be a tedious task. For instance, our binary model (M1) reaches an acceptable AUC of 0.69 with only 100 files. However, still, the model performs better with more files (*e.g.*, an AUC of 0.82 for 200 files). That said, developers do not need to deal with all the files in a project to classify them before using our model, developers can instead train a first model M1 with an acceptable AUC, then use that model to identify additional configuration and non-configuration files that are the TP and FP files of M1. Using a first model is more likely to identify configuration files than not using anything and randomly guessing which files are for configuration. Such a manual and random labeling is equivalent to an AUC of 0.50, while our model trained on just 100 files can have an AUC of 0.69 and of course, the more files used to train a model, the better the performance is. In the following, we provide use case scenarios of our models for both developers and researchers.

Models use case scenarios: Models use case scenarios: The identification and classification of configuration files, particularly in large systems, can provide an initial starting point for both developers and researchers to gain a better understanding of the configuration system. On one hand, using our classification models will help developers identify the different types of configuration files of OpenStack. Eventually, developers can use their understanding of the various types of files as a foundation to understand their roles, and knowing which types of files need to change for a configuration-related modification. For instance, if a developer decides to remove a configuration option of a service from the "External" configuration file, but forgets to remove the option's reference from the "Declaration" file, the service would still attempt to use the removed option from the declaration file upon startup, and eventually may fail to restart or behave unexpectedly. Furthermore, knowing the types of configuration files will provide a better understanding of the high level picture of configuration files. For instance, the "Infrastructure-Templates" are often used to generate the "External" configuration files (cf. RQ1 section). Therefore, in case a developer decides to manually modify the auto-generated external file, the changes would be lost once the external file is decided to be removed or re-auto generated. For example, the external "nova.conf" file[38] is auto-generated from Puppet. While it is possible to manually modify it, it is not recommended to do so, as indicated in a comment of the same file: "*# HEADER: This file was autogenerated at Thu Oct 04 14:53:15 -0400 2012 by puppet. While it can still be managed manually, it is definitely not recommended.*"

On the other hand, using our models will help researchers analyze a configuration system from different perspectives. For example, researchers can (1) focus on certain types of configuration files that are most important to their research, (2) identify the interactions between configuration files to examine dependencies over different configuration, (3) provide further tools to suggest which configuration files need to co-change, and (4) better quantify the impact of configuration changes by considering all different types of configuration files in a system, etc. For example, the work of Jiang and Adams [27] only focuses on the evolution and maintenance of a single type of configuration files (files with ".pp" extension), which might underestimate the effort of maintaining the configuration system (*i.e.*, set of files that are used for the deployment and runtime configuration). Therefore, we encourage future works to use our classification model to determine different configuration types and consider all the types of configuration files, so managers can better plan their configuration-related changes.

[38]https://github.com/openstack/osops/blob/master/example-configs/MIT_CSAIL/controller/etc/nova/nova.conf

5 THREATS TO VALIDITY

In this section, we discuss the threats that might affect the validity of our results.

Construct threats to validity: Our first threat to validity concerns the selection of our manually studied configuration files. Since our qualitatively studied configuration files were selected by randomly sampling a set of files, our results might miss on some important types of configuration files. To mitigate this threat, we collect data using different sources, *i.e.*, the documentation of OpenStack, the documentation of the IaC tools used by OpenStack, and prior configuration related changes. In addition, we collected statistically representative samples for each of our three sources to end up with a large number (over 1,700) of configuration files. That said, our main take-home message will not change with more types of files. In fact, we conclude that a software configuration can have multiple types of files rather than just one file, so even covering more types of configuration files will not change our main take-home message.

Internal threats to validity: Our internal threat to validity consists of the impact of our selected training datasets on the performances of our two models. Our first classification model is trained on our manually labeled files and other randomly selected non-configuration files. Leveraging other configuration files or non-configuration files would lead to a different performance. To mitigate this risk, we extensively evaluated our models using 100 different bootstrap samples to make our conclusion statistically robust. Similarly, our second model leverages our manually classified configuration files, so adding new configuration files to our training dataset might change the results. Similarly to our first model, we mitigate this risk by leveraging 100 different bootstrap samples to make sure that our models' performances are consistent, and our good performances are not obtained just by chance.

Furthermore, an internal threat to validity is related to the process for selecting non-configuration files to train our classification model of RQ2. While our experiment uses the 1,756 files of RQ1 as configuration files, we followed a semi-automated approach to identify a corpus of non-configuration files. These non-configuration files can accidentally have configuration files, which can bias the trained classification model of RQ2. To mitigate this risk, we first eliminate all the files that were identified in our data selection process of configuration files (RQ1) and not just the 1,756 files that we manually classified. We then defined a set of keywords that should not appear in the path of a non-configuration file. The first and second authors iteratively refined these keywords using three iterations and 100 different randomly selected files (a total of 300 files). In addition, we eliminate any file that at least one rater (the first or second author) classified as a configuration file.

Another internal threat to validity is related to the distribution of configuration and non-configuration files. Since we do not know in advance what is the distribution of the two types of files in OpenStack, we evaluated our model using an equal distribution of files. However, such a decision can bias our model. To mitigate this risk, we evaluated whether the performance of our model for classifying configuration from non-configuration files is consistently performing on different sampling sizes (ranging from 50%/50% to 95%/5% with a 5% increment on the non-configuration files category). With that, we do not observe any significant differences between the models trained and tested under different sample distributions.

Furthermore, our effort analysis for the model that classifies configuration files under different types shows that one can obtain a high performance from a small sample of configuration files. An internal threat to validity concerns the number of configuration types that are obtained in that small sample. One might not end up selecting all the files, so our model will not be able to classify configuration files under these missed types. To mitigate this risk, we identified what are the chances to miss one or multiple configuration files when selecting a small sample of files. We observe that such a probability is low and the larger the sample is, the lower the probability of missing configuration types. We also encourage future studies to improve our model to better identify the different types of configuration files.

External threats to validity: A large body of prior work [27, 45, 80, 89] that focused just on OpenStack and our paper share the same threat of generalizing our results to other software systems. For instance, certain files that we identified can be unique to OpenStack, while other files that exist in other systems might not be identified through our study. That is, OpenStack uses several IaC tools that are popular, so some identified types of configuration files can still exist in other systems. Our conclusion is that it is worth exploring what files are used for configuration in a system ahead of improving configuration for that system, as different files can exist. We also advocate that machine learning classifiers can help in identifying configuration from non-configuration files and different types of configuration files. As we focus just on OpenStack configuration files, OpenStack is still a highly configurable software system and has many projects that were extensively studied in the literature. In addition, our study considers many qualitatively analyzed files. Similarly, our models are not generalizable for other software systems in other domains (*e.g.*, robotics, ML pipelines, operating systems, etc.). For example, ML pipelines have configurations for training, tuning and testing models [3], an operating system can have certain files/tools to define the constraints between different configuration options, other files for enabling/disabling certain drivers. While our results do not generalize to these files, our take-home message can be valid for these systems that can also have multiple files for configuration. Thus, we encourage future studies to replicate our work on these systems. That said, our results show that such a number of files can be as low as 100 files to reach a decent median AUC (*i.e.*, 0.69) for OpenStack. Although we do not generalize that using a specific sample of size will lead to the same performances that we obtained, we observe promising results when using a small subset of configuration files to train a machine-learning model. That can motivate future studies to replicate our experiment on other software systems. We also advocate practitioners and researchers first to evaluate a model with small sample size and incrementally increase that size to reach a good model's performance.

6 RELATED WORK

Most of the literature on software configuration focuses on empirically studying software configuration [24, 33, 62, 88] and debugging configuration errors [2, 64, 83, 84], while the closest work to our paper is related to the maintenance of software configuration [12, 15, 27, 37, 59, 65–67, 69, 70].

6.1 Empirical Studies on Software Configuration

A large body of research (*e.g.*, [24, 33, 62, 88]) investigated the complexity that configuration adds to the usability and maintenance of software systems. For example, Xu et al. [88] indicated that the increased complexity of software configuration can be simplified by removing unused configuration options and simplifying complex ones. Sayagh and Adams [62] empirically examined how the configuration options of different WordPress and its plugins are interrelated. Behrang et al. [4] presented the SCIC static analysis technique that can automatically detect software configuration inconsistencies in systems that are written in multiple programming languages and have complex configuration options structures. Chen et al. [10] studied software configuration dependencies in 16 widely-used cloud and datacenter systems to define and identify common code patterns for five types of configuration dependencies. The study shows that configuration dependencies are more prevalent and diverse and should be considered a first-class issue in software configuration engineering. Ramachandran et al. [60] introduced techniques that do not require a system administrator to have in-depth knowledge about a multi-tiered system in order to be aware of the possible configuration dependencies in a system. Hubaux et al. [24] conducted two surveys among Linux and sCos users to understand the configuration challenges. Jin et al. [33] analyzed a highly-configurable industrial application and two open source applications to quantify the true challenges that configurability creates for software testing and debugging. Guerriero et al. [21] shed light on the current state of practice in the adoption of IaC and the key software engineering challenges in the field. The paper aims to investigate how practitioners adopt and develop IaC, and what are the practitioner's needs when dealing with

IaC development, maintenance, and evolution. The study conducted 44 semi-structured interviews with senior developers from different companies. The findings of the study suggest that there is a need for more research in the field of IaC. The currently available IaC tools have limited support, and developers feel the need for novel techniques for testing and maintaining IaC code. The study highlights the challenges faced by practitioners in adopting and developing IaC, including the need for better support for testing and maintenance of IaC code. However, none of these studies investigated what constitute a configuration system, neither the challenges faced on different types of configuration files.

6.2 Debugging Configuration Errors

A large body of research (*e.g.*, [2, 64, 83, 84]) leveraged source code analysis techniques to help practitioners identify misconfigured options. Sayagh et al. [64] leveraged a modular source code analysis approach to identify misconfigured configuration options in the LAMP (Linux, Apache, MySQL, and Php) stack. Attariyan and Flinn [2] built an approach called ConfAid, which finds the root cause of configuration errors. Wang et al. [83] and Wang et al. [84] detected potential configuration errors in a machine exhibiting erroneous behavior by comparing its state to other machines. While these papers focus on software configuration, none of them studied what constitute a configuration system or the impact of each configuration type on the quality of a software system.

6.3 The Maintenance of Software Configuration

A large body of research undertaken the maintenance of software configuration [12, 15, 27, 37, 59, 65–67, 69, 70]. Shambaugh et al. [69] developed an approach called *Rehearsal* to check the non-deterministic *Puppet* IaC files and ensure the correct execution of scripts in different environments. Other researchers proposed code smells detectors for Puppet [59, 70], Chef [67], and Ansible [15] infrastructure-as-code configuration files. For instance, Sharma et al. [70] proposed a catalog of 13 implementations and 11 design configuration smells applied to Puppet IaC files. Rahman et al. [59] leveraged a tool that detects seven security-related code smells of Puppet. Similarly, Schwarz et al. [67] investigated code smells for Chef code by identifying possible violations of Chef design guidelines. In the same line, Furthermore, Vassallo et al. [82] provided a semantic linter "CD-LINTER" for the identification of smells in Continuous Delivery (CD) pipelines configuration files in 5,312 open-source projects on GitLab. Furthermore, Sayagh et al. [65] identified a catalog gathering nine major configuration activities, 22 configuration-related challenges, and 24 best practices to improve software configuration quality. Sayagh et al. [66] proposed the *Config2Code* framework for developers to better manage their configuration options and cope with software configuration challenges, including configuration maintenance. Kumara et al. [37] performed a large-scale observational analysis to gather good and poor IaC development practices for three IaC languages, namely Ansible, Puppet, and Chef. The authors provided a taxonomy consisting of ten good IaC practices and four bad practices. Jiang and Adams [27] conducted an empirical study on 265 OpenStack projects to investigate the co-evolution of IaC files (Puppet) and other files (production, test, and build files). They found that IaC files evolve as frequently as production and test files and require as much maintenance. Similarly, Cito et al. [12] examined the maintenance and evolution of Dockerfiles on a data set of over 70k instances. They found that Dockerfiles belonging to popular software projects are revised, with yearly averages ranging from five to 12 revisions. Furthermore, Ibrahim et al. [25] investigated the evolution of Docker Compose files based on their change history. They found that the Docker Compose files infrequently change, with a median number of only three commits. In the same line, Vassallo et al. [82] developed a semantic linter called "CD-LINTER" to detect 4 types of smells in configuration files of Continuous Delivery (CD) pipelines across 5,312 open-source projects on GitLab. The linter achieved a precision and recall of over 87%, indicating its effectiveness. The research also revealed that these identified issues are significant in real-world scenarios and occur frequently.

While we do not carry out a study on the maintenance of configuration files, our results encourage future studies to investigate the maintenance of the whole configuration system rather than just certain configuration files. For example, we encourage future studies to investigate code smells on different configuration files rather than just Puppet, Chef, and Ansible files.

7 CONCLUSION

While several studies investigated the maintenance of software configuration, they focused on a limited subset of the configuration system. To cope with such a limitation, we investigate what types can constitute a configuration system and whether we can leverage machine learning techniques to identify a configuration system and its different type of files. We qualitatively studied 1,756 configuration files to better understand what types of configuration files can constitute a configuration system. As a result, we report nine different configuration types used at the OpenStack deployment and run-time stages. These files are interconnected, as we observe that some configuration files used during deployment can generate other configuration files used in the run-time stage. Furthermore, these configuration files do not all use the same format or have the same file extension(s). We find that seven file extensions are used for configuration and non-configuration files, making identifying the different configuration file types a not straightforward task.

To help practitioners and researchers identify the different types of files, we propose in this paper two machine learning classification models. The first model identifies configuration from non-configuration files, and the second model classify a given configuration file under its appropriate type of configuration files. Both of our models leverage the textual content of files using the TF-IDF algorithm. Our first model is able to identify configuration from non-configuration files, with a median AUC of 0.91, a median Brier score of 0.12, and a median precision of 0.86, and recall of 0.83. We also find that our models can reach a median AUC of 0.69 from only 100 files to manually label. Our second model to classify the different types of configuration files shows a median weighted AUC of 0.92, a median Brier score of 0.04, and a median weighted precision or 0.84, and a median weighted recall of 0.82. Our second model only requires a set of 100 OpenStack configuration files to reach a weighted AUC higher than 0.75. Our results conclude that a software system can have multiple types of configuration files, whereas one can train a model with a minimal set of files to identify configuration files and classify them under different types. While we found promising results on OpenStack, we encourage future work to replicate our study on other projects.

ACKNOWLEDGMENTS

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), grants RGPIN-2018-05960 and RGPIN-2021-04000.

REFERENCES

[1] Paul Anderson. 2006. System Configuration. https://homepages.inf.ed.ac.uk/dcspaul/home/master/pdf/sage-sysconfig.pdf.
[2] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis.. In *OSDI*, Vol. 10. 1–14.
[3] Amine Barrak, Ellis E Eghan, and Bram Adams. 2021. On the co-evolution of ml pipelines and source code-empirical study of dvc projects. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 422–433.
[4] Farnaz Behrang, Myra B Cohen, and Alessandro Orso. 2015. Users beware: Preference inconsistencies ahead. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 295–306.
[5] Andrew P Bradley. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern recognition* 30, 7 (1997), 1145–1159.
[6] Glenn W Brier et al. 1950. Verification of forecasts expressed in terms of probability. *Monthly weather review* 78, 1 (1950), 1–3.
[7] Yevgeniy Brikman. 2019. *Terraform: Up & Running: Writing Infrastructure as Code.* O'Reilly Media.

[8] Denis Eka Cahyani and Irene Patasik. 2021. Performance comparison of TF-IDF and Word2Vec models for emotion text classification. *Bulletin of Electrical Engineering and Informatics* 10, 5 (2021), 2780–2788.

[9] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[10] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and discovering software configuration dependencies in cloud and datacenter systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 362–374.

[11] Yao-Tsung Chen and Meng Chang Chen. 2011. Using chi-square statistics to measure similarities for text categorization. *Expert systems with applications* 38, 4 (2011), 3085–3090.

[12] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. 2017. An empirical analysis of the docker container ecosystem on github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 323–333.

[13] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin* 114, 3 (1993), 494.

[14] William Jay Conover. 1998. *Practical nonparametric statistics*. Vol. 350. John Wiley & Sons.

[15] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. 2020. Towards a catalogue of software quality metrics for infrastructure code. *Journal of Systems and Software* (2020), 110726.

[16] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. 2021. Within-Project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics. *IEEE Transactions on Software Engineering* (2021).

[17] Danilo Dessi, Rim Helaoui, Vivek Kumar, Diego Reforgiato Recupero, and Daniele Riboni. 2021. TF-IDF vs word embeddings for morbidity identification in clinical notes: An initial study. *arXiv preprint arXiv:2105.09632* (2021).

[18] Kevin K Dobbin, Yingdong Zhao, and Richard M Simon. 2008. How large a training set is needed to develop a classifier for microarray data? *Clinical Cancer Research* 14, 1 (2008), 108–114.

[19] Facebook Engineering. 2021. Update about the October 4th outage. https://engineering.fb.com/2021/10/04/networking-traffic/outage/.

[20] Mikel Galar, Alberto Fernández, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera. 2011. An overview of ensemble methods for binary classifiers in multi-class problems: Experimental study on one-vs-one and one-vs-all schemes. *Pattern Recognition* 44, 8 (2011), 1761–1776.

[21] Michele Guerriero, Martin Garriga, Damian A Tamburri, and Fabio Palomba. 2019. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 580–589.

[22] Noriko Hanakawa and Masaki Obana. 2019. A Computer System Quality metric for Infrastructure with Configuration Files' Changes. In *Proceedings of the 2nd International Conference on Software Engineering and Information Management*. 39–43.

[23] Lorin Hochstein and Rene Moser. 2017. *Ansible: Up and Running: Automating configuration management and deployment the easy way. "* O'Reilly Media, Inc.".

[24] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 2012. A user survey of configuration challenges in Linux and eCos. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. 149–155.

[25] Md Hasan Ibrahim, Mohammed Sayagh, and Ahmed E Hassan. 2021. A study of how Docker Compose is used to compose multi-component systems. *Empirical Software Engineering* 26, 6 (2021), 1–27.

[26] EG Jelihovschi, José Cláudio Faria, and Ivan Bezerra Allaman. 2014. The ScottKnott clustering algorithm. *Universidade Estadual de Santa Cruz-UESC, Ilheus, Bahia, Brasil* (2014).

[27] Yujuan Jiang and Bram Adams. 2015. Co-evolution of infrastructure and source code-an empirical study. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 45–55.

[28] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Ahmed E Hassan. 2018. The impact of correlated metrics on defect models. *arXiv preprint arXiv:1801.10271* (2018).

[29] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Ahmed E Hassan. 2019. The impact of correlated metrics on the interpretation of defect models. *IEEE Transactions on Software Engineering* (2019).

[30] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Akinori Ihara, and Kenichi Matsumoto. 2016. A study of redundant metrics in defect prediction datasets. In *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 51–52.

[31] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Christoph Treude. 2020. The impact of automated feature selection techniques on the interpretation of defect models. *Empirical Software Engineering* 25, 5 (2020), 3590–3638.

[32] Dongpu Jin, Myra B Cohen, Xiao Qu, and Brian Robinson. 2014. PrefFinder: getting the right preference in configurable software systems. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 151–162.

[33] Dongpu Jin, Xiao Qu, Myra B Cohen, and Brian Robinson. 2014. Configurations everywhere: Implications for testing and debugging in practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*. 215–224.

[34] Xin Jin, Anbang Xu, Rongfang Bie, and Ping Guo. 2006. Machine learning techniques and chi-square feature selection for cancer classification using SAGE gene expression profiles. In *International workshop on data mining for biomedical applications*. Springer, 106–115.

[35] HM Kalayeh and David A Landgrebe. 1983. Predicting the required number of training samples. *IEEE transactions on pattern analysis and machine intelligence* 6 (1983), 664–667.
[36] Klaus Krippendorff. 2018. *Content analysis: An introduction to its methodology.* Sage publications.
[37] Indika Kumara, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Damian Andrew Tamburri, Willem-Jan van den Heuvel, and Fabio Palomba. 2021. The do's and don'ts of infrastructure code: A systematic grey literature review. *Information and Software Technology* (2021), 106593.
[38] Daniel Lee, Gopi Krishnan Rajbahadur, Dayi Lin, Mohammed Sayagh, Cor-Paul Bezemer, and Ahmed E Hassan. 2020. An empirical study of the characteristics of popular Minecraft mods. *Empirical Software Engineering* 25, 5 (2020), 3396–3429.
[39] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E Hassan. 2020. A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering* (2020).
[40] Jialiang Li and Jason P Fine. 2010. Weighted area under the receiver operating characteristic curve and its application to gene selection. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 59, 4 (2010), 673–692.
[41] Mary L McHugh. 2013. The chi-square test of independence. *Biochemia medica: Biochemia medica* 23, 2 (2013), 143–149.
[42] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189.
[43] Davoud Davoudi Moghaddam, Omid Rahmati, Mahdi Panahi, John Tiefenbacher, Hamid Darabi, Ali Haghizadeh, Ali Torabi Haghighi, Omid Asadi Nalivan, and Dieu Tien Bui. 2020. The effect of sample size on different machine learning models for groundwater potential mapping in mountain bedrock aquifers. *Catena* 187 (2020), 104421.
[44] Sayan Mukherjee, Pablo Tamayo, Simon Rogers, Ryan Rifkin, Anna Engle, Colin Campbell, Todd R Golub, and Jill P Mesirov. 2003. Estimating dataset size requirements for classifying DNA microarray data. *Journal of computational biology* 10, 2 (2003), 119–142.
[45] Tomonobu Niwa, Yuki Kasuya, and Takeshi Kitahara. 2017. Anomaly detection for openstack services with process-related topological analysis. In *2017 13th International Conference on Network and Service Management (CNSM)*. IEEE, 1–5.
[46] Nur Nurmuliani, Didar Zowghi, and Susan P Williams. 2004. Using card sorting technique to classify requirements change. In *Proceedings. 12th IEEE International Requirements Engineering Conference, 2004.* IEEE, 240–248.
[47] OpenStack. 2010. Example of declaration configuration file used at run-time of OpenStack. https://github.com/openstack/nova/blob/master/nova/conf/api.py. (Accessed on 14/1/2022).
[48] OpenStack. 2011. Example of infrastructure-setup configuration file used during deployment of OpenStack. https://github.com/openstack/puppet-nova/blob/master/manifests/init.pp. (Accessed on 8/3/2023).
[49] OpenStack. 2012. Example of reader configuration file used at run-time of OpenStack. https://github.com/openstack/cinder/blob/master/cinder/api/api_utils.py. (Accessed on 14/1/2022).
[50] OpenStack. 2013. Example of resource-operations configuration file used at run-time of OpenStack. https://github.com/openstack/aodh/blob/master/devstack/plugin.sh. (Accessed on 14/1/2022).
[51] OpenStack. 2014. Example of infrastructure-setup configuration file used during deployment of OpenStack. https://github.com/openstack/kolla-ansible/blob/master/ansible/roles/keystone/tasks/config.yml. (Accessed on 14/1/2022).
[52] OpenStack. 2014. Example of infrastructure-templates configuration file used during deployment of OpenStack. https://github.com/openstack/kolla-ansible/blob/master/ansible/roles/keystone/templates/keystone.conf.j2. (Accessed on 14/1/2022).
[53] OpenStack. 2014. Example of infrastructure-variables configuration file used during deployment of OpenStack. https://github.com/openstack/kolla-ansible/blob/master/ansible/group_vars/all.yml. (Accessed on 14/1/2022).
[54] OpenStack. 2019. Example of environment-access configuration file used at run-time of OpenStack. https://docs.openstack.org/python-openstackclient/pike/configuration/index.html. (Accessed on 14/1/2022).
[55] OpenStack. 2019. Example of infrastructure-creation configuration file used during deployment of OpenStack. https://github.com/openstack/loci/blob/master/scripts/setup_pip.sh. (Accessed on 14/1/2022).
[56] OpenStack. 2021. Example of external configuration file used during deployment of OpenStack. https://docs.openstack.org/nova/queens/configuration/sample-config.html. (Accessed on 14/1/2022).
[57] Package:. 2022. https://github.com/stilab-ets/CongIdentification.
[58] Armanda Eka Putra, Luh Kesuma Wardhani, et al. 2019. Chi-Square Feature Selection Effect On Naive Bayes Classifier Algorithm Performance For Sentiment Analysis Document. In *2019 7th International Conference on Cyber and IT Service Management (CITSM)*, Vol. 7. IEEE, 1–7.
[59] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 164–175.
[60] Vinod Ramachandran, Manish Gupta, Manish Sethi, and Soudip Roy Chowdhury. 2009. Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications. In *Proceedings of the 6th international conference on Autonomic computing*. 169–178.
[61] Abdul Rafiez Abdul Raziff, Md Nasir Sulaiman, Norwati Mustapha, and Thinagaran Perumal. 2017. Single classifier, OvO, OvA and RCC multiclass classification method in handheld based smartphone gait identification. In *AIP Conference Proceedings*, Vol. 1891. AIP

Publishing LLC, 020009.
[62] Mohammed Sayagh and Bram Adams. 2015. Multi-layer software configuration: Empirical study on wordpress. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 31–40.
[63] Mohammed Sayagh and Ahmed E Hassan. 2020. ConfigMiner: Identifying the Appropriate Configuration Options for Config-related User Questions by Mining Online Forums. *IEEE Transactions on Software Engineering* (2020).
[64] Mohammed Sayagh, Noureddine Kerzazi, and Bram Adams. 2017. On cross-stack configuration errors. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 255–265.
[65] Mohammed Sayagh, Noureddine Kerzazi, Bram Adams, and Fabio Petrillo. 2018. Software configuration engineering in practice interviews, survey, and systematic literature review. *IEEE Transactions on Software Engineering* 46, 6 (2018), 646–673.
[66] Mohammed Sayagh, Noureddine Kerzazi, Fabio Petrillo, Khalil Bennani, and Bram Adams. 2020. What should your run-time configuration framework do to help developers? *Empirical Software Engineering* 25, 2 (2020), 1259–1293.
[67] Julian Schwarz, Andreas Steffens, and Horst Lichter. 2018. Code smells in infrastructure as code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 220–228.
[68] CSO: Cloud security configuration errors put data at risk; new tools can help. 2017. System Configuration. https://www.csoonline.com/article/3251605/cloud-security-configuration-errors-put-data-at-risk-new-tools-can-help.html.
[69] Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 416–430.
[70] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 189–200.
[71] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.
[72] Chakkrit Tantithamthavorn and Ahmed E Hassan. 2018. An experience report on defect modelling in practice: Pitfalls and challenges. In *Proceedings of the 40th International conference on software engineering: Software engineering in practice*. 286–295.
[73] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1200–1219.
[74] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. 2015. The impact of mislabelling on the performance and interpretation of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 812–823.
[75] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering*. 321–332.
[76] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Comments on "Researcher bias: the use of machine learning in software defect prediction". *IEEE Transactions on Software Engineering* 42, 11 (2016), 1092–1094.
[77] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2016), 1–18.
[78] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering* 45, 7 (2018), 683–711.
[79] Mischa Taylor and Seth Vargo. 2014. *Learning Chef: A Guide to Configuration Management and Automation*. " O'Reilly Media, Inc.".
[80] Jia Tong, Li Ying, Yuan Xiaoyong, Tang Hongyan, and Wu Zhonghai. 2015. Characterizing and predicting bug assignment in openstack. In *2015 Second International Conference on Trustworthy Systems and Their Applications*. IEEE, 16–23.
[81] James Turnbull and Jeffrey McCune. 2011. *Pro Puppet*. Vol. 1. Springer.
[82] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C Gall, and Massimiliano Di Penta. 2020. Configuration smells in continuous delivery pipelines: a linter and a six-month study on GitLab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 327–337.
[83] Helen J Wang, John C Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. 2004. Automatic Misconfiguration Troubleshooting with PeerPressure.. In *OSDI*, Vol. 4. 245–257.
[84] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J Wang, Chun Yuan, and Zheng Zhang. 2004. Strider: A black-box, state-based approach to change and configuration management and support. *Science of Computer Programming* 53, 2 (2004), 143–164.
[85] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2020. Predicting defective lines using a model-agnostic technique. *arXiv preprint arXiv:2009.03612* (2020).
[86] Wei Wen, Tingting Yu, and Jane Huffman Hayes. 2016. Colua: Automatically predicting configuration bug reports and extracting configuration options. In *2016 IEEE 27Th international symposium on software reliability engineering (ISSRE)*. IEEE, 150–161.
[87] Bowen Xu, David Lo, Xin Xia, Ashish Sureka, and Shanping Li. 2015. Efspredictor: Predicting configuration bugs with ensemble feature selection. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 206–213.
[88] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint*

Meeting on Foundations of Software Engineering. 307–319.
[89] Yoji Yamato, Shinichiro Katsuragi, Shinji Nagao, and Norihiro Miura. 2015. Software maintenance evaluation of agile software development method based on OpenStack. *IEICE TRANSACTIONS on Information and Systems* 98, 7 (2015), 1377–1380.
[90] Feng Zhang, Ahmed E Hassan, Shane McIntosh, and Ying Zou. 2016. The use of summation to aggregate software metrics hinders the performance of defect prediction models. *IEEE Transactions on Software Engineering* 43, 5 (2016), 476–491.
[91] Thomas Zimmermann. 2016. Card-sorting: From text to themes. In *Perspectives on data science for software engineering*. Elsevier, 137–141.