



Towards understanding code review practices for infrastructure-as-code: An empirical study on OpenStack projects

Narjes Bessghaier¹ · Ali Ouni¹ · Mohammed Sayagh¹ · Moataz Chouchen¹ · Mohamed Wiem Mkaouer²

Accepted: 30 March 2025

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2025

Abstract

Infrastructure-as-code (IaC) is a widely used practice to automate the creation, provisioning, orchestration, and configuration of infrastructures (such as networks, databases, and services). IaC allows the use of specification files in the form of code to manage the infrastructure. Practitioners can apply quality assurance best practices like code review to evolve IaC artifacts like any other software system. Code review (CR) is a common practice in which developers (*i.e.*, reviewers) review code changes submitted by their peers to fix errors and ensure that the code adheres to standards. Previous studies have shown that MCR can improve the overall quality of the code, and that MCR practices may vary depending on the code under review. Yet, little is known about how MCR practices are used in the context of IaC compared to Non-IaC code. While Non-IaC focuses on implementing system components, IaC translates these functional requirements into configuration code that defines infrastructure behavior, highlighting the need to explore their distinct review practices. This paper presents the first empirical study to investigate how practitioners perform code reviews for IaC code changes. Using a dataset of over 300k code reviews from the OpenStack ecosystem, we found that both IaC and Non-IaC code changes take a comparable merge time. However, IaC developers make 1.82 times more churn, while reviewers exchange 1.1 times more messages when reviewing IaC-related code changes. We further examined the contribution of experienced reviewers in reviewing IaC code changes. Our findings reveal that the top 5% of reviewers participate in 44% to 75% of IaC code changes, indicating that dedicated reviewers are assigned to review IaC code changes in OpenStack. Finally, to gain a better understanding of the factors influencing reviewers in the evaluation of IaC code changes, we conducted a rigorous thematic analysis on a representative sample of IaC code changes. We created a checklist with seven main topics and 38 sub-topics through a thematic analysis. To validate our checklist, we surveyed nine developers amongst the most active OpenStack reviewers in IaC code changes. This approach strengthens the reliability of our findings and adds empirical support to the identified checklist. This latter can be useful as a foundation of IaC guidelines for developers along the IaC code change development and review process to check the quality of their IaC code changes. In conclusion, we emphasize the importance of recognizing the non-trivial nature of the IaC code change review. We argue for increased attention from researchers

Communicated by: Jeffrey C. Carver

Extended author information available on the last page of the article

Published online: 26 April 2025

Springer
Content courtesy of Springer Nature, terms of use apply. Rights reserved.

to explore other dimensions of IaC code changes, recognizing that these are indispensable practices for ensuring the robustness of software systems infrastructure.

Keywords Infrastructure-as-code · Code review · Code changes · Thematic analysis

1 Introduction

Infrastructure-as-Code (IaC) is an emerging practice to continuously configure software systems infrastructure, such as installing packages, creating user profiles, managing permissions, and installing services (Redhat 2022; Jiang and Adams 2015). IaC is supported by multiple tools, such as Puppet (Turnbull and McCune 2011), Chef (Taylor and Vargo 2014), Ansible (Hochstein and Moser 2017), and Terraform (Brikman 2019), each offering different provisioning and customization capabilities. These IaC tools enable practitioners to use machine-readable specification files to manage infrastructure in the same way as software application code (Guerriero et al. 2019). By using IaC, practitioners can version control their infrastructure-related code and benefit from development tools and practices that have improved the efficiency of software development processes (Rahman et al. 2019). That is, IaC adoption has grown in modern software projects, particularly in both commercial and open-source projects.¹ Recently, several companies, including IBM² and Microsoft,³ started to adopt the IaC technology into their development cycles to set up their infrastructures.

While IaC brings several advantages, it also comes with several challenges. Using various IaC tools can lead to an increased code complexity as it involves managing many infrastructure components and dependencies (Moris 2021; Guerriero et al. 2019; Kula et al. 2018). As stated by Gene et al. *"Infrastructure as Code practices have evolved from managing servers to managing complete stacks, but this new power comes at the cost of complexity"* (Kim et al. 2016; Moris 2021). Furthermore, in the constantly evolving nature of IaC code to configure and scale infrastructures, misconfiguration issues often emerge and can have a widespread impact (Rahman et al. 2020). Hence, several projects use quality assurance practices, such as code review, to maintain the quality of IaC code changes similar to any other software artifacts. Code review is a common quality assurance practice that consists of manually inspecting code to identify potential quality issues and ensure the adherence of the code to the best practices (Bacchelli and Bird 2013; Sadowski et al. 2018).

Several studies have investigated code review practices in source code (Bacchelli and Bird 2013; Sadowski et al. 2018; McIntosh et al. 2016; AlOmar et al. 2022; Coelho et al. 2021). However, little is known about IaC code review practices and challenges. Unlike coding with traditional programming languages, IaC relies on machine-readable definition files to code provisioning and deployment processes instead of doing so manually. We speculate that reviewing IaC-related code changes have distinct characteristics and challenges than Non-IaC-related code changes. This distinction primarily arises from the pivotal role of IaC code, which can significantly influence the overall system infrastructure by introducing changes to interconnected components that must function harmoniously and seamlessly (Bessghaier et al. 2023). Furthermore, the integration of various IaC technologies for managing the system's infrastructure introduces further complexity, resulting in the development of diverse

¹ <https://octoverse.github.com>

² <https://www.ibm.com/cloud/learn/infrastructure-as-code>

³ <https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/infrastructure-as-code>

IaC artifacts using different paradigms.⁴ While code review practices for IaC may share common aspects as general-purpose programming languages (GPLs), such as ensuring code quality, identifying bugs, and improving maintainability, there are several key differences due to the nature and purpose of IaC compared to production and test code. IaC is not traditional code but rather configuration code that defines, provisions, and manages infrastructure including some factors that are not present in GPL. IaC focuses on translating operational requirements into configuration scripts that define infrastructure behaviour. Developers must ensure different aspects such as correctness in configurations, idempotency, security best practices, resource optimization, and compatibility with the diverse environments (e.g., staging, production) (Bessghaier et al. 2023; Rahman and Parnin 2023). In contrast, GPL code review focuses on business logics, source code performance, maintainability, and adherence to software design principles. Second, errors and defects in IaC can lead to critical infrastructure failures, such as downtime, security breaches, or resource misallocation, affecting the stability and reliability of the whole software system (Xu and Zhou 2015). GPL errors, while also impactful, are found to impact different aspects of the systems such as security, performance, etc (Ray et al. 2014). Third, IaC requires detailed and up-to-date documentation of infrastructure setup, dependencies, and configurations that link different components as well as external providers. This differs from GPL code, where documentation often centres on API usage and code logics. Finally, the growing reliance on IaC makes understanding its code review practices crucial to improving software infrastructure reliability. Hence, the particularity of IaC code may trigger different challenges for developers that we aim to explore and understand in this paper.

This motivates the investigation of the IaC code review effort and the type of reviewers involved with the IaC code review. By investigating IaC code changes, we can potentially advance the understanding of IaC artifacts to improve IaC development practices. Understanding the review practices used in IaC code changes can help managers more effectively plan for these reviews and allow developers to improve the quality of their IaC code changes by following best practices.

In this paper, we aim to understand the review process for IaC scripts. In particular, we conduct a three-fold empirical study to (1) quantitatively compare code review attributes (such as duration, number of reviewers, code churn, and number of revisions) adopted for IaC code changes (where at least one IaC file is included in the change) to that of Non-IaC code changes (where no IaC file is included in the change) using the Wilcoxon rank-sum non-parametric test (Conover 1998) and Cliff delta (Cliff 1993) to examine the attribute's statistical difference, (2) examine the contribution of reviewers concerning IaC and Non-IaC code changes, and (3) qualitatively identify the criteria that developers discuss before accepting an IaC code change by conducting a thematic analysis (Glaser and Strauss 1967; AlOmar et al. 2024) and validate the checklist through surveying OpenStack reviewers. We conduct our study on the OpenStack ecosystem, which leverages various IaC tools to configure and deploy infrastructures. Specifically, OpenStack has followed a six-month release cycle (OpenStack 2019) with three phases (development, release-candidate, and post-release) since the early 2012 release (Diablo⁵). This provided an opportunity to longitudinally examine how the code review attributes of IaC differ across the release cycle compared to Non-IaC code by collecting a dataset of over 300k merged code changes. This provides a comprehensive view of the effort devoted to reviewing IaC code changes, particularly throughout the different

⁴ <https://betterstack.com/community/comparisons/chef-vs-puppet-vs-ansible/>

⁵ <https://www.openstack.org/blog/openstack-announces-diablo-release/>

phases of the OpenStack release cycle. It reveals the types of issues that IaC developers commonly encounter in every phase of the cycle. This analysis helps highlight key challenges and patterns in the IaC review process, contributing to a deeper understanding of the overall effort involved. Our study is guided by the following research questions:

- **RQ1: How often do developers review IaC code changes?** Our findings uncovered that IaC code changes, representing 10% of all code changes, are submitted for review in the three phases of the OpenStack release cycle. Additionally, we found that IaC code changes follow the same evolution pattern as Non-IaC code changes in the 12 OpenStack releases.
- **RQ2: How different are code review practices when reviewing IaC and Non-IaC code changes?** We found that during the development phase, IaC code changes take longer to be reviewed than Non-IaC code changes (an average of 1.2 times longer) and involve more reviewers (a median of five for IaC and four for Non-IaC) with a negligible effect size. However, we also observe that during the release-candidate and post-release phases, the IaC code changes related to bugs are reviewed at a faster pace compared to code changes related to Non-IaC bugs, even though the number of assigned reviewers is the same for both code changes (a median of three in the release-candidate phase and a median of four in the post-release phase). Additionally, we found that during an IaC code change, reviewers tend to exchange a higher number of messages, particularly during the development phase (a median of 15 for IaC compared to a median of 13 for Non-IaC) and the release-candidate phase (a median of nine for IaC compared to a median of eight for Non-IaC) with a negligible effect size.
- **RQ3: What is the contribution of reviewers working on IaC code changes?** Our observations show that IaC reviewers are contributing more to the review process of OpenStack compared to those working only on Non-IaC code changes. Furthermore, we found that the top-5% of reviewers participate in a minimum of 301 IaC code changes (i.e., 44% of the changes) to a maximum of 693 (i.e., 75% of the changes) code changes across the 12 OpenStack releases. Furthermore, we found that the top 5% of reviewers are authoring more IaC code changes (i.e., 60% of the changes) than the remaining 95% of reviewers (i.e., 40% of the changes). This suggests that the developers who are most active in reviewing IaC code changes are also heavily involved in developing those changes.
- **RQ4: What criteria are considered during the code review to merge IaC code changes?** Through a manual analysis of a representative sample of 379 merged IaC code changes, and a survey with nine OpenStack most active reviewers, we identified and validated seven topics that were discussed by the reviewers and code changes owners, covering a total of 38 specific issues related to IaC coding, such as “*Set deprecated parameters to UNDEF*”, which ensures that deprecated options no longer have a value. These issues can constitute a code review checklist to ensure proper coding of IaC code changes.

The results of our empirical study advocate that the review effort of IaC code changes, in terms of code review attributes, is comparable to that of Non-IaC code changes. Consequently, we advocate researchers and developers to be more aware of the importance of reviewing IaC scripts. It is noteworthy that the existing literature lacks comprehensive studies delving into IaC code review, and to the best of our knowledge, there are no established quality models or guidelines specific to IaC. Furthermore, understanding the process of code review and the specific criteria used to evaluate IaC code changes is important for both developers

and researchers, as it can help ensure that code changes are of high quality and conform to standards and best practices. Code review is a critical step in the development process. It is important to ensure that it is done effectively, especially for code changes related to infrastructure, as mistakes or vulnerabilities in IaC code can have serious consequences. Therefore, following best practices when writing code changes in IaC can help increase the chances that the code will be accepted and merged. Some best practices for writing IaC code include adhering to the DSL annotations and built-in functions, commenting code to explain its purpose, and testing the IaC code changes before submitting them for review. Adhering to these best practices can make it easier for reviewers to understand and evaluate the code. It is also important for developers to be responsive to feedback from reviewers and to make any necessary changes or corrections on time to avoid delays in the code review process.

Data Availability Statement The datasets generated during and/or analyzed during the current study and the scripts are available in the following GitHub repository, <https://github.com/stilab-ets/iacreview>.

Paper organization The rest of the paper is organized as follows. Section 2 presents background on IaC tools and code review and provides a motivating example. Section 3 describes our study design, while Section 4 presents and discusses the quantitative and qualitative analysis results. Section 5 highlights the implications of our study. Section 6 discusses our threats to validity. Section 7 reflects on some studies on the practices of IaC and code review. Finally, we conclude the paper and discuss our future research directions in section 8.

2 Background and Motivation

In this section, we first provide an overview of infrastructure-as-code (IaC) concepts, code review, and the OpenStack release cycle. We then motivate our study goals with a real IaC code change example.

2.1 Background

2.1.1 Infrastructure-as-code

Infrastructure-as-code (IaC) allows the automatic management and provisioning of infrastructures using machine-readable specification files. One of the main features of IaC is the use of code to describe the infrastructure, as well as the ability to version control and automate the configuration of the infrastructure. IaC often includes tools for deployment, provisioning, and management, such as Puppet (Turnbull and McCune 2011), Chef (Taylor and Vargo 2014), Ansible (Hochstein and Moser 2017), and Terraform (Brikman 2019). Several tools are available for use with IaC, and each can be used for specific purposes in infrastructure configuration. For example, Terraform can be used to create a cloud environment or set up a firewall. In contrast, tools like Ansible or Puppet can be used to install resources such as a web server onto the configured infrastructure. Among the IaC tools, Puppet and Ansible are particularly popular and widely used by many companies (such as Atlassian⁶ and Intel⁷).

⁶ <https://www.atlassian.com/microservices/cloud-computing/infrastructure-as-code>

⁷ <https://www.intel.la/content/www/xl/es/financial-services-it/cloud/building-a-unified-pipeline.html>

We provide, in Listings 1 and 2, snippets of configuration tasks written in Puppet and Ansible, respectively. In the Puppet code, a domain-specific language is employed to define and manage four resources on the host server. Specifically, Puppet ensures the presence of `package1`, creates a user named `user1`, and generates a configuration file (`conf_file`) containing the content “hello” and ensures that the `service1` service is running. On the contrary, in the Ansible code, YAML syntax is utilized to articulate equivalent configuration tasks. For instance, the `Apt` module⁸ is used to manage packages and ensures that `package1` is present. The `Copy` module⁹ is used to ensure that the file at “`src/config/conf_file`” exists and has the content “hello”. The `User` module¹⁰ is employed to manage user accounts. In this example, it ensures that the user named `user1` is present and has a home directory managed. Finally, the `Systemd` module¹¹ is used to manage services and ensures that `service1` is running and enabled.

Listing 1 Puppet code

```

1  Package {'package1':
2      ensure => present, }
3  File {'src/config/conf_file':
4      ensure => file,
5      content => 'hello' }
6  User {'user1':
7      ensure => present,
8      managehome => true }
9  Service {'service1':
10     ensure => running,
11     enable => true }
```

Listing 2 Ansible code

```

1  - name: set 'package1'
2      apt:
3          name: package1
4          state: present
5  - name: create 'src/config/conf_file'
6      copy:
7          content: 'hello'
8          dest: src/config/conf_file
9  - name: set 'user1'
10     user:
11         name: user1
12         state: present
13         createhome: yes
14  - name: set 'service1'
15     systemd:
16         name: service1
17         state: started
18         enabled: yes
```

While IaC brings many benefits, there are also challenges to consider when adopting it Guerriero et al. (2019). One major challenge is the need for developers to learn new skills and adapt to new workflow processes. As pointed out by Guerriero et al. (2019), we quote:

⁸ https://docs.ansible.com/ansible/latest/collections/ansible/builtin/apt_module.html

⁹ https://docs.ansible.com/ansible/latest/collections/ansible/builtin/copy_module.html

¹⁰ https://docs.ansible.com/ansible/latest/collections/ansible/builtin/user_module.html

¹¹ https://docs.ansible.com/ansible/latest/collections/ansible/builtin/systemd_module.html

“infrastructure triaging involves many different formats which are often obscure to most and need specialized personnel.”; i.e., IaC tools come in different programming languages and paradigms that may not be easily interpreted by developers who lack expertise in this domain. There may also be technical challenges with integrating IaC tools with existing systems, so it is important for developers to carefully consider how IaC will fit into their existing workflow and to be mindful of any potential vulnerabilities that may be introduced (Guerriero et al. 2019).

2.1.2 Code Review

Developers often use various quality assurance practices, such as *code review* (Bacchelli and Bird 2013), to maintain the quality of their code. Code review is the process of manually reviewing code changes to fix errors and ensure that the code meets certain standards (McIntosh et al. 2016; Baum et al. 2016; Peruma et al. 2020; AlOmar et al. 2022; Alrubaye et al. 2019). Modern Code review (MCR) is a tool-based approach that uses platforms like *Gerrit*¹² to facilitate an interactive and asynchronous code review process (Bacchelli and Bird 2013; Davila and Nunes 2021; Thongtanunam et al. 2017). During MCR, code changes are discussed between the owner of the change and expert developers to improve the code and determine whether it should be merged or abandoned (Yang et al. 2016). The code review process typically includes the following steps:

1. *Code change submission*: The author prepares and submits a code change (a set of code changes) to the code review tool;
2. *Reviewers' invitations*: After the code change is submitted, reviewers are invited to provide feedback. Reviewers can be invited by the author, recommended by a tool based on their expertise, or join the review voluntarily;¹³
3. *Code change review*: The reviewer(s) carefully examine the code change and provide comments, voting on whether the code change requires further revisions or is ready to be merged;
4. *Address reviewers' feedback*: The author makes revisions to the code change based on the reviewers' comments. There may be multiple versions of the originally submitted code change (called “patch-sets”) during this process;
5. *Code change status update*: Once the review process is complete, a committer (usually an experienced developer) updates the status of the code change based on the outcome of the review process, either merging the code change or abandoning it.

2.1.3 OpenStack Release Cycle

In our study, we focus on the OpenStack ecosystem, which is a widely-used open-source cloud computing platform. OpenStack is a set of software tools for building and managing cloud computing infrastructures. It operates by orchestrating compute, storage, and networking resources through a set of interrelated services. Additionally, OpenStack employs dedicated IaC projects to automate the deployment and management of the services. Furthermore, OpenStack has a complex interconnected nature (involving many contributors), its incorporation of new configuration technologies, and its active development cycle (six-month release

¹² <https://www.gerritcodereview.com>

¹³ https://docs.openstack.org/contributors/en_GB/code-and-documentation/using-gerrit.html

cycle), which has been extensively used in several studies in software engineering (AlOmar et al. 2022; Teixeira and Karsten 2019; Han et al. 2021; Thongtanunam et al. 2017; Li et al. 2022). Firstly, by examining OpenStack, we aim to investigate the IaC code issues faced by developers within a well-engineered and mature ecosystem. OpenStack's complexity and sophistication offer a rich source of data on the challenges and best practices in IaC development, allowing us to derive meaningful and actionable insights. Secondly, focusing on OpenStack helps us avoid the noise and inconsistencies that might arise from manually studying disparate IaC-based projects. OpenStack's cohesive and interconnected nature ensures that our findings are relevant and generalizable. This minimizes the risk of concluding projects that do not accurately reflect common practices or issues in IaC development. Thirdly, OpenStack offers an opportunity to study the evolution of the IaC code review process over time. The continuous development and integration cycles within OpenStack enable us to track and analyze how IaC practices evolve and how review processes adapt. This longitudinal perspective is crucial for understanding the dynamics of IaC code review and its role in the software development lifecycle. Additionally, an OpenStack release goes through three main phases (*development*, *release-candidate*, and *post-release*) (Teixeira and Karsten 2019; OpenStack 2021), which allows us to longitudinally compare IaC and Non-IaC code changes throughout the evolution of the release cycle and see how IaC code changes differ from one phase to the next. Our goal is to gain a better understanding of IaC code review within the context of a complex ecosystem throughout the different stages of the release cycle. Firstly, it helps us understand the effort devoted to IaC code changes by revealing when the most effort is expended and why, providing insights into the workload associated with IaC code changes during each phase of the release cycle. Secondly, the release cycle provides a chronological framework to analyze the review process, identifying critical periods when review efforts peak, whether during the development phase for adding new features, during the testing phase of the release cycle, or during the post-release phase for fixing defects. Finally, the release cycle helps us map the types of issues encountered during code reviews to specific phases. This phased approach offers a nuanced understanding of the challenges IaC developers face along a project release cycle. As shown in Fig. 1, an OpenStack release cycle consists of the following three phases:

1. **Development:** The first phase of an OpenStack release cycle is the development period, which lasts for at least five months and includes activities such as planning, implementation, testing, and bug fixes (OpenStack 2016a, b). After three months, the development phase focuses on bug-fixing and feature freeze. This means that the development efforts

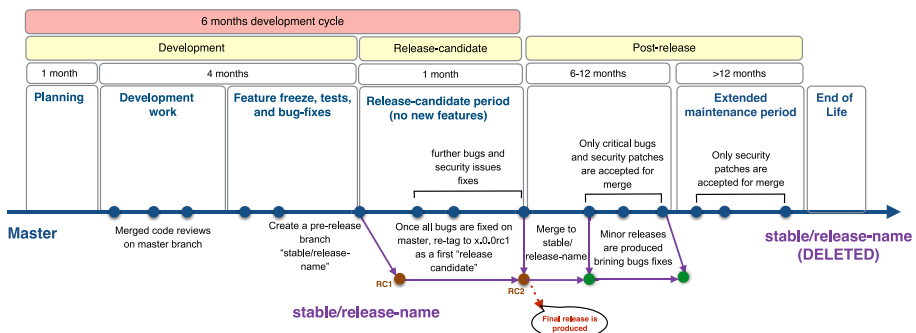


Fig. 1 An overview of the OpenStack release cycle

become centered on fixing bugs and stabilizing the system, and only changes that fix bugs and do not introduce new features can be merged into the master branch (OpenStack 2021).

2. **Release-candidate:** Once all identified critical bugs have been fixed, developers create a stable branch called “RC1” from the current state of the master branch, which will hold the first release candidate of the OpenStack release (OpenStack 2016a). During this phase, which is typically one month long (OpenStack 2019), only critical bug fixes and exceptional features (known as Feature Freeze Exceptions or FFEs) are allowed to be merged into the stable branch (OpenStack 2021). The RC1 may be used as is, to be released as the final OpenStack release unless new critical bugs are discovered. If new critical bugs are found, they must be merged into the master branch before being included in the stable branch, and a new RC2 is created (OpenStack 2022, 2016b). This process is repeated until all critical bugs have been fixed, and the final release date is reached (OpenStack 2021). Non-critical bugs may be documented as “*known issues*” in the release notes and addressed later, to avoid disruptions (Teixeira and Karsten 2019). On the final release date, the final RC from each project is collected and integrated to create the new OpenStack release for the cycle (OpenStack 2021).
3. **Post-release:** After the OpenStack release is produced, the post-release phase begins. During this phase, maintenance activities are carried out on the project’s final RC to fix known bugs and address any new bugs that are discovered after the projects have been deployed by clients (OpenStack 2022, 2016a).

2.2 Motivating Example

We provide in Fig. 2 an example of a code review (ID:303562)¹⁴ from the *tripleo-heat-templates* project¹⁵ of OpenStack using the Gerrit code review platform. The code change includes 10 existing files with two newly added files (“*services/neutron-l3.yaml*” and “*pacemaker/neutron-l3.yaml*”). The code change was submitted by “Dan Prince” to be reviewed by eight reviewers, along with the CI server (“Jenkins”). The code change pertains to an IaC-related code change, which adds a new puppet service. The review process took roughly 40 days and underwent 30 revisions before being merged. During the review process, the reviewers exchanged eight messages and left 13 inline comments to suggest improvements to the code. Following the reviewer’s directions, overall, 122 lines of code were added to or removed from the modified files. The reviewers discussed basic syntax issues (such as the leading comma “,” at the beginning of a line), wrong parameters values (such as, `neutron::agents::l3::enabled: false` and `neutron::agents::l3::manage_service: false`), and misplaced code (such as, the parameter

“`OS::TripleO::Services::NeutronL3Agent::OS::Heat::None`” should be under the “`resource_registry:`” section and not the “`parameter_defaults:`” section), which the reviewer “James Slagle” explicitly complained about it by saying: “*shouldn’t this be under the resource_registry and not parameter_defaults?*”. Furthermore, the reviewers pointed out that configuration parameters’ values should be updated if the current code change gets merged before another code change, which presumably could lead to configuration dependency issues if not done. Besides, a reviewer acknowledged a critical

¹⁴ <https://review.opendev.org/c/openstack/tripleo-heat-templates/+303562>

¹⁵ <https://github.com/openstack/tripleo-heat-templates>

The screenshot shows a Gerrit code review interface. At the top, the OpenDev logo and navigation links (CHANGES, DOCUMENTATION, BROWSE) are visible. The change is titled 'composable neutron l3 service' and is in a 'Merged' state. The 'Change Info' section on the left lists the submitter (May 18, 2016), owner (Dan Prince), and reviewers (Jenkins, Emilien Macchi, Steven Hardy, Brent Eagles, Giulio Fidente, James Slagle). The 'Repo | Branch' is 'openstack/tripleo-heat-templates | master' and the 'Topic' is 'bp/composable-services-within-roles'. The 'Submit requirements' section shows that the change has passed Code-Review (+2 from Steven Hardy, +2 from Emilien Macchi), Verified (+2 from Jenkins), and Workflow (+1 from Emilien Macchi). The 'Links' section includes a 'gitea' link. The 'Change-Id' is 'I0316043efe357a41ef3b4088a55d98dbb6d25963'. The 'Comments' section shows 13 resolved comments. The 'Files' section lists the files changed, including 'environments/neutron-midonet.yaml', 'environments/neutron-nuage-config.yaml', 'environments/neutron-opencontrail.yaml', 'environments/neutron-plumgrid.yaml', 'environments/puppet-pacemaker.yaml', 'overcloud.yaml', 'overcloud-resource-registry-puppet.yaml', 'puppet/controller.yaml', 'puppet/manifests/overcloud_controller.pp', 'puppet/manifests/overcloud_controller_pacemaker.pp', 'puppet/services/neutron-l3.yaml' (Added), and 'puppet/services/pacemaker/neutron-l3.yaml' (Added). The 'Files' table shows the size and delta for each file, with a total delta of +37 lines and -15 lines. The 'Commit message' is visible at the top of the file list.

| File | Comments | Size | Delta |
|--|-----------|------|--------|
| Commit message | | | |
| environments/neutron-midonet.yaml | | II | +1 -1 |
| environments/neutron-nuage-config.yaml | | II | +1 -1 |
| environments/neutron-opencontrail.yaml | | II | +1 -1 |
| environments/neutron-plumgrid.yaml | | III | +3 -3 |
| environments/puppet-pacemaker.yaml | | I | +1 -0 |
| overcloud.yaml | | II | +1 -5 |
| overcloud-resource-registry-puppet.yaml | | I | +1 -0 |
| puppet/controller.yaml | | II | +0 -12 |
| puppet/manifests/overcloud_controller.pp | | I | +0 -5 |
| puppet/manifests/overcloud_controller_pacemaker.pp | | II | +0 -15 |
| puppet/services/neutron-l3.yaml | | II | +37 -0 |
| puppet/services/pacemaker/neutron-l3.yaml | 1 comment | II | +33 -0 |

Fig. 2 An example of a code review (ID:303562) from the Tripleo-heat-templates project of OpenStack using Gerrit

oversight, stating that they did not realize or recall that certain configuration settings were removed, as we quote “*I was confused cos on the side of that you only added to a few services, didn’t notice/remember that on the puppet-tripleo side you removed it for all the things*”. This highlights a common issue where developers sometimes lose track of modifications made to the code or may not be aware of important configuration changes that need to take place. Finally, the reviewers have voted a “+2” to accept the code change and merge it into the code base. This example motivates the importance of comprehending the issues that reviewers raise during IaC code changes, as well as examining whether experienced reviewers are involved in these code change reviews.

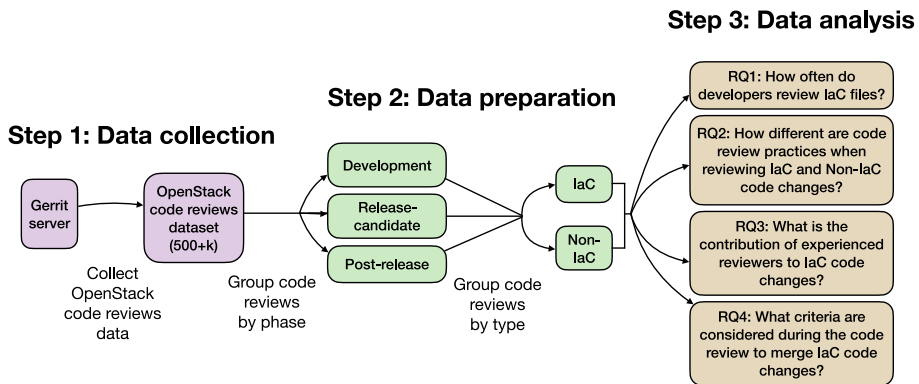


Fig. 3 Empirical study overview

3 Empirical Study Design

The main focus of our study is to investigate code review practices in the context of IaC. To do so, we first quantitatively analyze IaC code changes using various code review attributes, such as the number of revisions, the number of reviewers, the review duration, etc. Furthermore, we dig deeper to examine the contribution of reviewers to the IaC code review practice. Then, we qualitatively examine the specific code issues that reviewers discuss when evaluating IaC code changes. As depicted in Fig. 3, our study consists of the data collection, data preparation, and data analysis steps.

3.1 Step 1: Data Collection

We mined the code review data using the REST API¹⁶ that Gerrit provides to query the code changes from opendev code review hosting platform,¹⁷ which returns the results in JSON format. We crawled all closed code changes with the status of either “Merged” or “Abandoned”. We collected a total of 554,672 code changes, from which we finally selected 365,407 merged code reviews. Overall, we mined the code reviews of 19 OpenStack releases (from Diablo (OpenStack 2011) to Victoria (OpenStack 2020)) having a code review lifetime from 2011 to 2021.

3.2 Step 2: Data Preparation

In our study, we aim to perform a fine-grained analysis by breaking down each release cycle into its three phases, *i.e.*, (1) development, (2) release-candidate, and (3) post-release phases (cf. Section 2.1.3). Hence, we analyze the collected code changes based on the cycle phase, and then we identify and group the code changes related to IaC and other code (*i.e.*, Non-IaC).

¹⁶ <https://gerrit-review.googlesource.com/Documentation/rest-api.html>

¹⁷ <https://review.opendev.org/q/status:open+-is:wip>

3.2.1 Step 2.1: Categorize Code Changes by Cycle Phase

Since the beginning of the development of OpenStack in 2010, the project has not followed a consistent release cycle. Some releases (such as Austin¹⁸ and Bexar¹⁹) came at irregular intervals, ranging from three to five months. As a result of this analysis, we removed seven OpenStack releases before 2015 (Diablo to Juno), where the release-candidate phase did not follow the consistent pattern of one to two months, and we ended up with 12 releases left (Kilo to Victoria).

To accurately determine the dates of the three phases of the 12 OpenStack releases, we consider the OpenStack release deadline and the stable branch creation dates for each release. The development phase of a release $R_{(i)}$ begins on the day after the last OpenStack release deadline ($R_{(i-1)} + 1$) and ends on the day the stable branch of $R_{(i)}$ is created (OpenStack 2021). Any code changes merged into the master branch during this period belong to the development phase. The release-candidate phase begins on the day the stable branch of release $R_{(i)}$ is created and ends on the final release date of $R_{(i)}$. The post-release phase of release $R_{(i)}$ starts the day after the release deadline ($R_{(i)} + 1$) and continues until the last code review date is merged into the stable branch in our dataset. Any code changes merged into the stable branch after the creation of the stable branch belong to either the release-candidate or post-release phase, depending on the time they were merged. In Table 1, we show the characteristics of the 12 studied OpenStack releases.

3.2.2 Step 2.2: Categorize Code Changes into IaC and Non-IaC

After classifying all code changes by phase, we need to distinguish IaC from Non-IaC code changes. OpenStack uses Puppet and Ansible IaC tools, which have been extensively studied in previous research (Sharma et al. 2016; Rahman et al. 2019; Van der Bent et al. 2018; Opdebeeck et al. 2022; Dalla Palma et al. 2020). Puppet uses dedicated files with the “.pp” extension to define all the resources that will be installed during deployment,²⁰ while Ansible defines its resources in YAML files under the “roles”, “tasks”, or “playbooks” folders.²¹ We wrote custom scripts to identify all code changes that contain at least one of these two types of IaC files. We ended up with IaC-related code changes accounting for 10% out of the total number of merged studied code changes.

For example, in Fig. 4, we provide an example of an IaC-related code change.²² The code change includes adding the developer mode to the project *Bifrost*. Thus, a configuration option called “developer_mode” is added to the source code python file *cli.py*²³ responsible for implementing Command-Line-Interface (CLI) commands for the Bifrost project. The option is then defined in the Ansible file *defaults/main.yml*²⁴ where options defaults are declared. Later, an Ansible task has been added to the Ansible IaC file *tasks/main.yml*,²⁵ so

¹⁸ <https://releases.openstack.org/austin/index.html>

¹⁹ <https://releases.openstack.org/bexar/index.html>

²⁰ https://www.puppet.com/docs/puppet/5.5/lang_summary.html

²¹ https://docs.ansible.com/ansible/2.8/user_guide/playbooks_best_practices.html

²² <https://review.opendev.org/c/openstack/bifrost/+744233>

²³ <https://github.com/openstack/bifrost/blob/master/bifrost/cli.py>

²⁴ <https://github.com/openstack/bifrost/blob/master/playbooks/roles/bifrost-pip-install/defaults/main.yml>

²⁵ <https://github.com/openstack/bifrost/blob/master/playbooks/roles/bifrost-pip-install/tasks/main.yml>

Table 1 OpenStack releases statistics, with (M) referring to merged code changes and (A) referring to abandoned code changes (CC)[illegible]

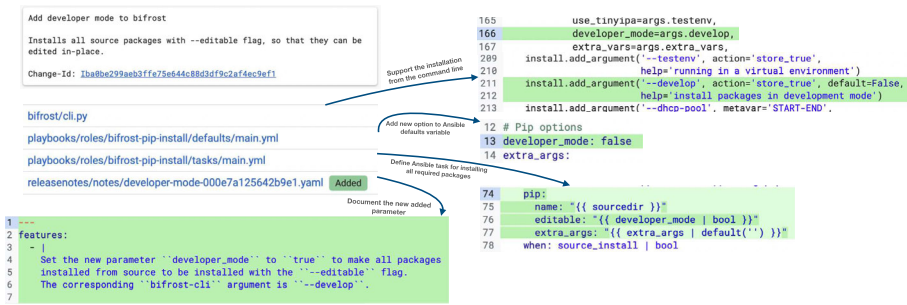


Fig. 4 Example of an IaC-related code review (ID:744233) of the Bifrost project containing Non-IaC scripts

that the new option takes effect in the environment. Finally, a release notes file²⁶ is added to document the change.

3.3 Step 3: Data Analysis

To address our research questions, we conduct a mixed analysis with both quantitative and qualitative methods.

3.3.1 Step 3.1: Quantitative Data Analysis

RQ1: How often do developers review IaC code changes? To answer RQ1, we count the weekly number of IaC and Non-IaC code changes to visualize the evolution of code changes in each phase along the release. Besides, given the limited release candidate phase time frame of up to two months, we aim to identify any notable peaks in IaC code changes in the weeks before the release date. Furthermore, we use the Spearman rank correlation between IaC and Non-IaC code changes for each release, to determine whether there is a statistically significant trend or pattern in the evolution of IaC and Non-IaC code changes over time and how these two types of code evolve with each other within the context of our study. The following are the intervals used to interpret the magnitude of the Spearman correlation:

- Negligible: between 0 and 0.20;
- Weak: between 0.21 and 0.40;
- Moderate: between 0.41 and 0.60;
- Strong: between 0.61 and 0.80;
- Very strong: between 0.81 and 1.

RQ2: How different are code review practices when reviewing IaC and Non-IaC code changes? Then, to answer RQ2 and similarly to AlOmar et al. (2021), we compare the code review practice for both IaC and Non-IaC code changes for all the releases combined using 10 different code review attributes as described in Table 2. Subsequently, we compare IaC and Non-IaC code changes for each phase (development, release-candidate, and post-release) to discern how each phase's particularity in code changes may impact the code review attributes. These latter cover various aspects of the code review process, such as the number of revisions, the number of changed files, the code churn, added and deleted lines

²⁶ <https://github.com/openstack/bifrost/blob/master/releasenotes/notes/developer-mode-000e7a125642b9e1.yaml>

Table 2 The 10 studied attributes of a code review process

| # | Attributes | Definition & Rationale |
|---|------------------------|---|
| 1 | No. of revisions | This attribute captures the committed changes required before a code change can be merged. High values of this attribute may suggest that it is challenging to revise IaC code changes and that they often require multiple iterations before being approved for merging. This information can provide insight into the difficulty of reviewing and revising IaC code changes. |
| 2 | No. of files | This attribute reflects the number of files that are modified together in a code change that is being reviewed. A high number of changed files may indicate that an IaC change has a broad impact on the codebase and may affect other files. This information can be useful in understanding the scope of an IaC code change and how it may impact the overall system. |
| 3 | Churn | The churn measures the amount of code that is added or deleted in a code change that is being reviewed. High churn values may suggest that an IaC code change is more difficult to maintain and may require additional modifications over time. This attribute may also indicate that significant changes are being requested within the same code review. Understanding the churn of IaC code changes can provide insight into the complexity and maintainability of these changes. |
| 4 | Added lines | This attribute captures the number of new lines of code added to a file in a code change being reviewed. This attribute can provide information on the extent to which a file has been modified as part of a code change. Understanding the number of added lines in an IaC code change can help gauge the change's scope and impact. |
| 5 | Deleted lines | This attribute captures the number of lines of code that are removed from a file in a code change being reviewed. Understanding the number of deleted lines in an IaC code change can help assess the scope of the change, particularly in terms of code simplification, refactoring, or removal of obsolete or redundant code. |
| 6 | Description length | The description length is an attribute of the number of words written in the description of a code change being reviewed. A long description may suggest that it is necessary to provide detailed information about the changes made to fully describe an IaC code change. This attribute can provide insight into the complexity of IaC code changes and the amount of information that is needed to accurately convey the nature of the change. |
| 7 | No. of messages | This attribute captures the number of messages that are exchanged between reviewers and the owner of a code change being reviewed during the review process. A high number of messages may indicate that reviewers are encountering challenges in reviewing or revising an IaC code change. This attribute can provide insight into the difficulty of reviewing IaC code changes and may help to identify areas where additional support or clarification may be needed. |
| 8 | No. of inline comments | This attribute captures the number of comments that reviewers include within a changed code file to address specific issues. Inline comments are used to provide specific guidance or clarification on the changes being made in a code review. Understanding the number of inline comments in an IaC code change can provide insight into the level of detail and focus that is being applied during the review process. This metric is normalized by the number of added and deleted lines. |
| 9 | Duration | This attribute captures the amount of time that elapses between the submission of a code review and its merging. The duration of a code review can provide insight into how long it takes for an IaC code change to be approved and incorporated into the codebase. Understanding the duration of IaC code reviews can help managers plan and prioritize their review process. |

Table 2 continued

| # | Attributes | Definition & Rationale |
|----|------------------|--|
| 10 | No. of reviewers | This attribute captures the number of people who are involved in reviewing a code change. A high number of reviewers indicates that IaC code changes require more input or scrutiny before they can be approved. Understanding the number of reviewers involved in reviewing IaC code changes can provide insight into the level of resources that may be required to review this type of change and may help managers plan and allocate review resources appropriately. |

of code, the code change description length, the number of exchanged messages during the review process, the number of inline comments, the code review duration and the number of reviewers.

As we compare various review attributes for both IaC and Non-IaC code changes, we need to assess, for each attribute, whether the variation is statistically significant. Therefore, we apply the Wilcoxon rank-sum test (Cuzick 1985) and the Cliff's Delta effect size (Cliff 1993). The Wilcoxon rank-sum test is a non-parametric statistical test that is used to compare two independent groups, and we used it with a confidence level of 95% (p-value < 0.05). The null hypothesis for this test is that there is no difference in the code review attributes between IaC and Non-IaC code changes. Additionally, we perform corrections to our p-values using the Bonferroni correction (Napierala 2012). Furthermore, we use Cliff's Delta non-parametric effect size to estimate the magnitude of differences between the IaC and Non-IaC code changes for each attribute. The following are the categories used to interpret the magnitude of the difference:

- Negligible: for $|\delta| < 0.147$;
- Small: for $0.147 \leq |\delta| < 0.33$;
- Medium: for $0.33 \leq |\delta| < 0.474$;
- Large: for $|\delta| \geq 0.474$.

RQ3: What is the contribution of reviewers working on IaC code changes? Finally, in RQ3, we further complement our IaC code changes analysis by comparing the contribution score of reviewers whose code changes incorporate IaC with those whose code changes do not involve IaC following the approach of previous studies (Peruma et al. 2019; AlOmar et al. 2021). We use the number of reviewed and authored code changes as a proxy to measure the reviewers' contribution compared to all code changes up to each OpenStack release date, allowing for comparability between all reviewers in a release. It is important to note that the reviewers' contribution, denotes how much reviewers are contributing to the review and development of code changes. It is distinct from the code review attributes discussed in RQ2, which encompass measures quantifying the effort involved in reviewing a code change, including factors like churn, the number of reviewers, duration, etc.

To ensure a fair and consistent assessment of each reviewer's contribution, we employ a quantification approach that involves evaluating their contributions on a release-by-release basis. As illustrated in Fig. 5, we have two reviewers A and B reviewing code changes during three releases (R1, R2, and R3). The contribution of both reviewers in R1, R2, and R3 is R1: {A=2}, R2: {A=3, B=2}, R3: {A=4, B=5}. Given the substantial volume of over 300k code reviews, it becomes impractical to treat each review as a distinct checkpoint. Therefore, by using release dates as checkpoints, we can effectively maintain a

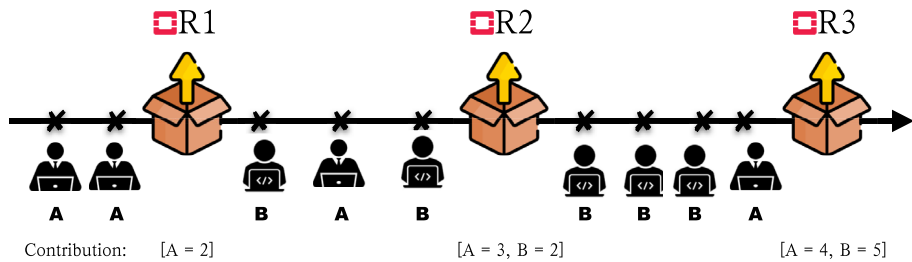


Fig. 5 Reviewer's contribution overtime

comprehensive evaluation and facilitate the comparability of each reviewer's contributions within the context of the OpenStack project's development during the 12 studied releases.

In our dataset, as shown in Table 3, we have 301 reviewers who review at least one IaC code change and 1,128 who review at least one Non-IaC code change. It is important to note that we combined reviewers exclusively working on only IaC code changes with those who review at least one IaC code change. This consolidation was deemed necessary due to the negligible size of the former group, as presented in Table 3.

Formally, we define the Reviewer Contribution Score (RCS) of each reviewer (R_i) by dividing their number of reviewed and authored code changes (CC) by the total number of code changes up to each OpenStack release date (V_j) as defined in (1):

$$RCS(R_i, V_j) = \frac{\sum (\text{Number of reviewed CC, Number of authored CC})_{R_i}}{\text{Total number of CC}_{(V_j)}} \quad (1)$$

Then, we investigate whether specific reviewers are significantly contributing to the review of IaC code changes, or whether it is randomly distributed among reviewers. To do so, we divide the reviewers into two groups for comparison. Similarly to AlOmar et al. (2021), the first group consists of reviewers that fall within the top 5% based on their RCS scores (that

Table 3 Reviewer's characteristics in the 12 studied, OpenStack releases code changes (referred to as "CC")

| Release | At least 1 IaC CC | At least 1 Non-IaC CC | Only IaC CC | Only Non-IaC CC | At least 1 IaC and 1 Non-IaC CC | # of distinct Reviewers | Top-5% |
|----------|-------------------|-----------------------|-------------|-----------------|---------------------------------|-------------------------|--------|
| Kilo | 301 | 2,773 | 30 | 2,502 | 271 | 2,803 | 140 |
| Liberty | 502 | 3,060 | 49 | 2,607 | 453 | 3,109 | 155 |
| Mitaka | 641 | 3,223 | 55 | 2,637 | 586 | 3,278 | 164 |
| Newton | 641 | 3,260 | 61 | 2,680 | 580 | 3,321 | 166 |
| Ocata | 601 | 2,761 | 52 | 2,212 | 549 | 2,813 | 141 |
| Pike | 620 | 2,647 | 39 | 2,066 | 581 | 2,686 | 134 |
| Queens | 693 | 2,310 | 44 | 1,661 | 649 | 2,354 | 118 |
| Rocky | 632 | 2,148 | 46 | 1,562 | 586 | 2,194 | 110 |
| Stein | 628 | 1,989 | 57 | 1,418 | 571 | 2,046 | 102 |
| Train | 484 | 1,704 | 37 | 1,257 | 447 | 1,741 | 87 |
| Ussuri | 434 | 1,418 | 36 | 1,020 | 398 | 1,454 | 73 |
| Victoria | 344 | 1,128 | 31 | 815 | 313 | 1,159 | 58 |

we denote as “top-5%”), and the second group contains the remaining 95% of reviewers. The top-5% group ranges from 58 to 166, whereas the remaining reviewers range from 1,101 to 3,209 for the studied releases. It is worth mentioning that we remove the bots from the list of reviewers by identifying the “tags”: [“SERVICE_USER”] from the “REVIEWER” element in the JSON files of Gerrit.

3.3.2 Step 3.2: Qualitative Data Analysis

RQ4: What criteria are considered during the code review to merge IaC code changes?

To answer RQ4, we applied the thematic analysis technique based on the guidelines of Cruzes and Dyba (2011) to create a taxonomy of the criteria that reviewers check while reviewing IaC code changes. This approach involves analyzing data to identify and develop themes (“topics”), within a collection of descriptive labels (“issues”) as a commonly used technique in software engineering (AlOmar et al. 2022; Silva et al. 2016). We derive our IaC code review criteria from a manual analysis of a representative sample of IaC code changes. To ensure objectivity during our analysis, the first two authors independently completed each step of the analysis, with their results cross-validated by all co-authors. We used the thematic analysis to perform several iterations of analyzing the IaC code changes. Our process for creating this taxonomy includes the following steps:

1. *Initial identification of labels:* Initial identification of labels: During this step, the first two authors independently analyze the code change owner and reviewers’ discussions and inline comments of IaC code changes to identify the discussed and resolved IaC coding issues explicitly stated by the reviewers. Each author will independently associate an issue with a label, otherwise, they associate it with an existing label. After identifying all IaC coding issues, all co-authors joined to discuss and refine the labels. In total, 66 IaC-related labels were identified. Among the identified labels, 37 labels were semantically equivalent and later standardized, such as “misplaced code” and “wrong code position”, with another eight being accepted and five causing conflicts. The remaining 16 labels were removed. All the authors involved in this study have experience in configuration and software engineering, ranging from three to 10 years.
2. *Reviewing labels for merging opportunities:* During this step, the labels identified during the first step were linked. This helped to identify and regroup related labels. Following discussion with the co-authors, four new labels were identified, 15 labels were merged (including the five conflicted labels from the first step), and the remaining 35 labels were relabeled. For example, as our analysis is not a single-step process, we identified a new issue related to “Adding workarounds in separate files”, and relabeled the issue “taking into consideration the execution pipeline” to “Check your configuration dependencies”.
3. *Translate labels into themes:* This step involves identifying the generic themes that describe the grouped labels generated in the second step. This process identified nine major topics discussed within the analyzed issues. As our analysis is iterative, we further refined our labels and topics. We merged two topics (“Code quality” and “IaC guidelines”) under the “Best-practices” topic and removed three labels.

It is essential to only consider those issues whose solutions are either explicitly stated as resolved by the code change owner or are evident in the revised version of the code (referred to as the code “DIFF”). This approach ensures that we only analyze IaC code changes that were accepted after the resolution of the issues stated by the reviewers. As we have many merged

Table 4 Professional experience of the participants

| Years of experience | Development(#) | OpenStack(#) | Code review(#) | IaC(#) |
|---------------------|----------------|--------------|----------------|--------|
| 4-6 | — | 3 | 1 | 4 |
| 7-9 | 2 | 2 | 2 | 2 |
| 10-12 | 1 | 4 | 4 | 3 |
| 13-16 | 3 | — | 2 | — |
| 17-19 | 1 | — | — | — |
| 20-23 | 2 | — | — | — |

with (#) referring to the number of participants

IaC code changes (308,657) in our OpenStack dataset, we randomly select a representative sample using the *Sampling* R package,²⁷ weighted by the distribution of the cycle phases (development, release-candidate, and post-release). The sample consists of 379 code changes that were selected with a 95% confidence level and a confidence interval of five (AlOmar et al. 2022). This implies that we are quite confident that the characteristics observed in our IaC code changes sample are indicative of the broader population of IaC code changes. The overall checklist building took approximately 20 days.

To validate our checklist, we reached out to the top 100 most active OpenStack reviewers, who frequently reviewed IaC code changes. Nine of the invited reviewers participated in our survey, resulting in an acceptable response rate of 9%, as per standards in software engineering (Khatoonabadi et al. 2023). Table 4 provides a summary of the reviewers' experience. Notably, 67% of the participants have seven to 23 years of overall software development experience, with eight to 10 years dedicated to the OpenStack ecosystem. Additionally, they bring 10 to 15 years of code review experience, coupled with six to 10 years of proficiency in IaC coding.

We developed a survey²⁸ using a 4-point Likert scale (Wang et al. 2023; Likert 1932) ("I disagree, I somewhat disagree, I somewhat agree, I agree). This binary approach (agree/disagree) was intended to ensure that the responses directly reflected the experts' views on the relevance of each item in our checklist, without the ambiguity that a neutral option might introduce. We conducted the survey among expert developers on IaC with the primary objective of validating the relevance of our checklist of IaC-related criteria. We aimed to ascertain whether each item on the checklist was perceived as relevant or not by these experienced professionals. By removing the neutral option, we encouraged respondents to take a definitive decision on each item, thereby providing us with clear indications of the checklist's validity. The survey comprised 26 questions, with the initial section focusing on five questions related to participant demographics. Subsequently, participants were presented with our checklist, consisting initially of seven topics and 36 items. For each topic, a multiple-choice question prompted respondents to rate the corresponding items using the Likert scale. Additionally, two short-answer questions were included, inviting participants to share their thoughts on the topic name and its practicality.

²⁷ <https://search.r-project.org/CRAN/refmans/samplingbook/html/pps.sampling.html>

²⁸ <https://docs.google.com/forms/d/1IUFE7fB5SAniRtGf2yc0EJNNg8QhkLqLEhMHgKDfsXM/edit#responses>

4 Empirical Study Results and Discussions

In this section, we report and discuss our findings for quantitatively and qualitatively analyzing code review practices when reviewing IaC-related code changes.

4.1 RQ1: How Often do Developers Review IaC Code Changes?

Motivation In this research question, we aim to understand how often reviewers deal with IaC code changes throughout the OpenStack release life cycle. Particularly, we aim to visualize any peaks of IaC code changes along the three phases and whether IaC and Non-IaC code changes follow the same pattern of evolution.

Approach To answer RQ1, we leverage our data to compare the number and evolution of weekly-basis reviewed IaC and Non-IaC code changes throughout the three cycle phases (development, release-candidate, and post-release) of the 12 OpenStack releases.

Results Our results reveal that despite the minority of IaC code changes representing only 10% of all code changes, they are reviewed during the whole life-cycle of an OpenStack release and are found statistically following the same pattern of evolution as the Non-IaC code changes.

Finding 1: We observe that developers review IaC code changes throughout the three phases of the OpenStack release cycle. In Fig. 6, we illustrate the evolution of the number of IaC and Non-IaC code changes through the three cycle phases (i.e., development, release-candidate, and post-release) of the 12 OpenStack releases per week. We observe that IaC code changes have increased from the first to the last release (Kilo to Victoria) indicating a maturation in the IaC codebase for OpenStack. As well, in three out of the 12 releases (i.e., Mitaka, Newton, and Ocata), there is a notable increase in the number of IaC code changes during the release-candidate phase. For instance, we have 149 IaC code changes reviewed in one week during the release-candidate phase of the Newton release. During the Newton release cycle, we found that OpenStack underwent a major change to add the interoperability

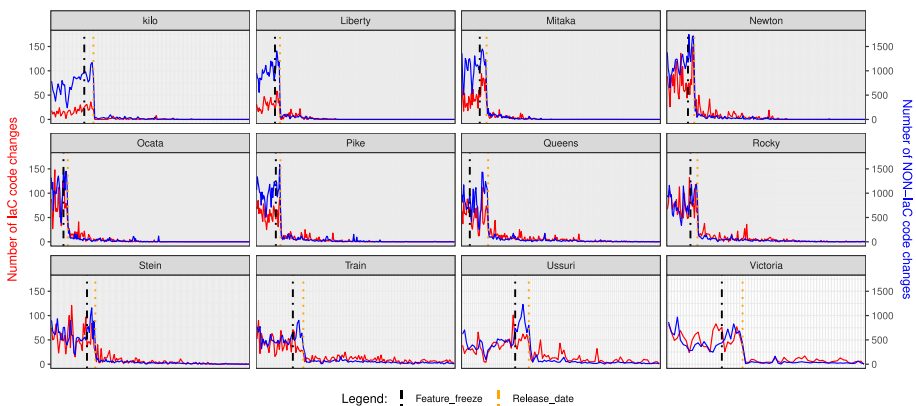


Fig. 6 Number of reviewed IaC and Non-IaC code changes per week throughout the 12 studied OpenStack releases. The black dotted line represents the end of the development phase and the start of the release-candidate phase. The orange dotted line represents the end of the release-candidate phase and the start of the post-release phase

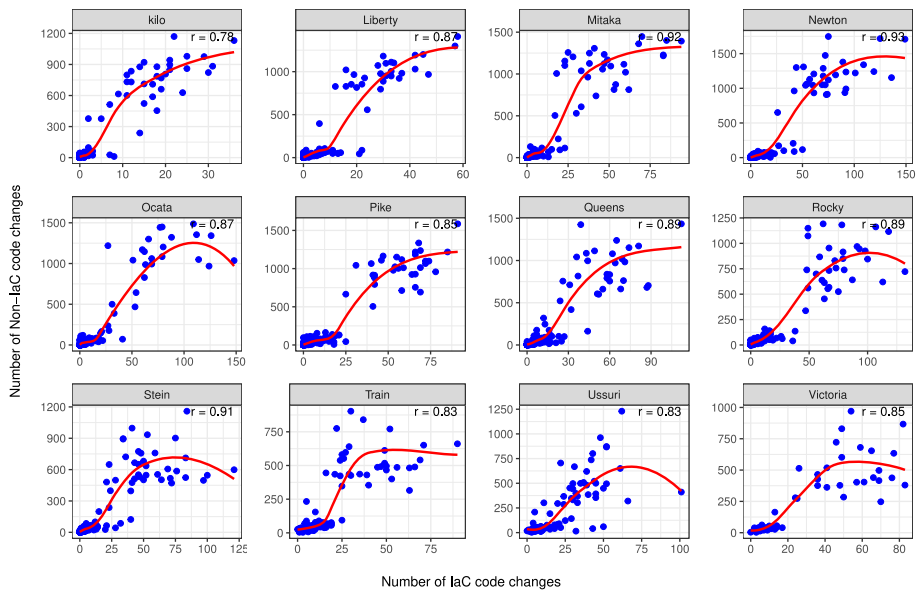


Fig. 7 Spearman correlation between IaC and Non-IaC code changes for the 12 releases

feature and better serve end users.²⁹ The observed peak of IaC code changes is likely a result of the extensive changes related to implementing the interoperability feature encompassing standards, APIs, and configurations as explained in the release notes.³⁰ This finding motivates us to further investigate how these IaC code changes are reviewed compared to Non-IaC code changes.

Finding 2: Our analysis reveals that IaC code changes follow the same evolution pattern as Non-IaC code changes across the three phases observed in the 12 OpenStack releases. As depicted in Fig. 7, our Spearman correlation shows a statistically significant positive relationship ($p\text{-value} < 0.001$) between IaC and Non-IaC code changes for the 12 releases. The correlation coefficient indicates that the Non-IaC code changes can effectively account for the variability in the IaC code changes, with a range spanning from 78% to 93%. These findings could indicate that the development activities in the three phases (i.e., development, release-candidate, and post-release) influence both IaC and Non-IaC code changes, which motivate us to further investigate, as future works, the dependency odds between both code changes. We emphasize that our correlation results do not imply causation between the evolution of IaC and Non-IaC code changes.

4.2 RQ2: How Different are Code Review Practices when Reviewing IaC and Non-IaC Code Changes?

Motivation The purpose of this study is to understand whether code review practices in IaC code changes differ from Non-IaC code changes. To this end, we use a set of code review attributes from previous research (AlOmar et al. 2021; Coelho et al. 2021) to compare code

²⁹ <https://docs.openstack.org/project-team-guide/introduction.html>

³⁰ <https://docs.openstack.org/releasenotes/neutron/newton.html>

reviews involving IaC scripts to Non-IaC code reviews. This analysis aims to understand how developers practice code review in the context of IaC code changes. Understanding such practice provides practitioners with insights on how to effectively manage the code review process in IaC code changes. This may include considerations such as the duration required to review an IaC change and the number of reviewers to assign to this type of change, etc.

Approach We compare the code review practices of IaC and Non-IaC code changes by leveraging 10 different code review attributes listed in Table 2. The attributes we analyzed include the number of revisions, number of reviewers, number of messages, duration, number of files, churn, number of added lines, number of deleted lines, description length, and number of inline comments. To further study if there is an impact of external factors on code review attributes, we perform Multiple Regression Analysis (MRA) (Edwards 1985) between the nine code review attributes and the confounding variables, namely the “size” and “age” of files in a code change, as well as the release “cycle” during which the review took place (i.e., development, release candidate, or post-release). We collect our confounding variables data as follows:

- *Size*: After collecting the code changes data for all OpenStack projects in JSON format (cf. Section 3.1), we extract the list of files associated with each code change using the node “files”. Next, we determine the size of each file by extracting the node “size” for every file within the respective code change.
- *Age*: For each file F_i in a code change C_i , we analyze the Git repository to identify the commit where the file was first introduced in the project. We then compute the file’s age by subtracting the date of creation of F_i in the project from the current date of C_i .
- *Cycle*: As discussed in Section 3.2.1, we classify code changes based on their cycle phase (development, release-candidate, or post-release). For each file F_i within C_i , we assign the corresponding cycle phase of C_i . For the Spearman correlation analysis, we numerically encode the cycle phases as follows: 1 for development, 2 for release-candidate, and 3 for post-release.

MRA helps identify the relationship between the confounding variables and the nine code changes attributes in terms of the impact of confounding variables on the variation of a code change attribute in IaC and Non-IaC code changes. In our study, we apply MRA to quantify the variation in the attributes, denoted as y_i , in terms of the effects of the confounding variables: “size”, “age”, and “cycle”, as expressed in (2)

$$y_i = \alpha + \beta_1 \cdot \text{Cycle}_i + \beta_2 \cdot \text{Age}_i + \beta_3 \cdot \text{Size}_i + \epsilon \quad (2)$$

We implement the MRA using the “lm” function from the `car`³¹ package in R. Similarly to Saidani et al. (2021), we apply a log transformation to the confounding variables (Cohen et al. 2013) to stabilize the variance and improve model fit. As well, to mitigate the risk of multicollinearity, where one predictor variable can be linearly predicted from the others (Cohen et al. 2013), we use the Variance Inflation Factor (package `car` in R). Following the same approach, we filter out the top 3% of the data as outliers in order to avoid inflating the model’s fit (Vasilescu et al. 2015). For each model, we report the following metrics: (i) *Coefficients*, which quantifies the mathematical relationship between each independent variable (confounding variable) and the dependent variable (code change attribute), with higher values indicating a stronger effect; (ii) *p-values*, which assesses the significance of the coefficients, providing insight into the reliability of each predictor; (iii) *Percentage of Sum of*

³¹ <https://cran.r-project.org/web/packages/car/car.pdf>

Table 5 Statistical significance between IaC and Non-IaC code changes review attributes

| Attributes | IaC | | Non-IaC | | Statistical Significance | | |
|------------------------|--------|--------|---------|--------|--------------------------|--------------------|--------------------|
| | Median | Mean | Median | Mean | p-value | Cliff d | Bonferroni p-value |
| No. of added lines | 13 | 70.37 | 7 | 90.75 | <2.2e-16 | Negligible (0.13) | <0.001 |
| No. of deleted lines | 3 | 41.34 | 2 | 64.25 | <2.2e-16 | Negligible (0.06) | <0.001 |
| Churn | 20 | 111.72 | 11 | 155 | <2.2e-16 | Negligible (0.13) | <0.001 |
| No. of revisions | 2 | 3.35 | 2 | 3.09 | <2.2e-16 | Negligible (0.03) | <0.001 |
| Description length | 40 | 41.02 | 39 | 40.13 | <2.2e-16 | Negligible (0.05) | <0.001 |
| No. of files | 2 | 4.74 | 1 | 2.86 | <2.2e-16 | Small (0.24) | 0 |
| No. of messages | 13 | 19.86 | 12 | 22.19 | <2.2e-16 | Negligible (0.05) | <0.001 |
| No. of inline comments | 0 | 2.07 | 0 | 3.20 | <2.2e-16 | Negligible (-0.02) | <0.001 |
| Duration (hours) | 73.42 | 380.23 | 73.32 | 438.08 | <2.2e-16 | Negligible (0.01) | <0.001 |
| No. of reviewers | 4 | 5.01 | 4 | 6.01 | 0.68 | Negligible (0.00) | 0.68 |

Squares, which represents the proportion of variance explained by each variable, offering a measure of its contribution to the overall model fit; and (iv) *Standard Error*, which indicates the variability of the regression model's predictions in the units of the response variable (code change attribute). Smaller standard error values suggest a more precise model fit and greater confidence in the estimates.”

Results Our results advocate that the code review process of IaC code changes could be as challenging as Non-IaC code changes, as captured by our 10 code review attributes. We list our main key findings as follows.

Finding 3: We observe that IaC code changes take as long as Non-IaC code changes, where developers perform a higher churn on more files with a negligible to small effect size. In Table 5, we provide an overview of the different code review attributes for IaC and Non-IaC code changes regardless of the release cycle phases. We first observe that IaC code changes undergo a higher churn, with a median of 20 compared to a median of 11 for Non-IaC code changes. This result might be expected, as a higher number of files are found to be included in IaC code changes with a median of two files compared to the Non-IaC with a median of one file. However, impacting a larger number of files makes it difficult for developers and reviewers to monitor the propagated change across the different files. For instance, in the IaC code change ID:391532,³² reviewers spotted that the file where the new change took effect is not appropriate for the change as one of the reviewers indicated: “*This seems like it should be done in nova_compute_kvm.yml rather than here*”. Furthermore, we found that in some IaC code changes, the reviewers request to refactor code into separate files, *i.e.*, adding new files, for better code clarity. For example, one of the reviewers commented, as part of reviewing the code change ID:391532:³³ “*Please refactor the new task into a separate file and import it in the proposed places.*” Consequently, the reviewers would exchange a higher number of messages to discuss all the changes, with a median of 13 compared to a median of 12 for the Non-IaC code changes, with a p-value < 2.2e-16 and a negligible effect size of 0.05. We found that IaC code changes take approximately as long as Non-IaC code

³² https://review.opendev.org/c/openstack/openstack-ansible-os_nova/+/391532

³³ <https://review.opendev.org/c/openstack/kolla-ansible/+/676219>

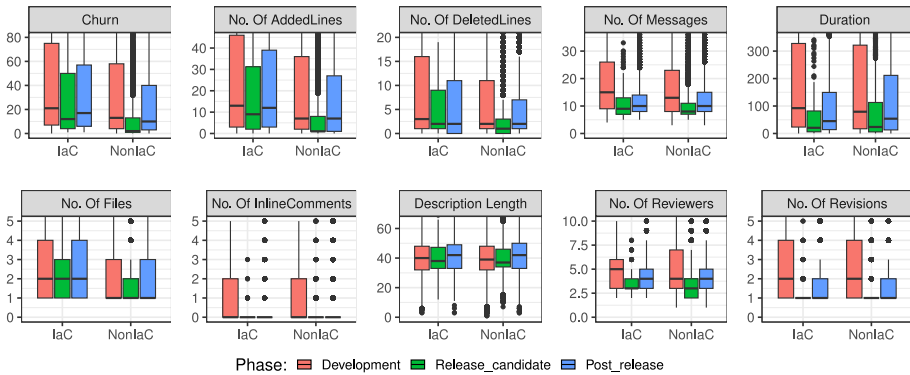


Fig. 8 Boxplots representing the distribution of the 10 code review attributes in the three cycle phases (development, release-candidate, and post-release)

changes to be reviewed, with a median of 73 hours with a p-value of $1.155e-05$ and a negligible effect size of 0.01. We also found that the same number of reviewers were assigned to both IaC and Non-IaC code changes (IaC and Non-IaC) with a median of four with no statistical difference (p-value of 0.68). However, in some IaC code changes, a reviewer might request the intervention of a puppet expert to validate a code change, as explicitly indicated in a review comment: “*Needs confirmation from puppet cores.*” in the code review ID:382551.³⁴ These findings suggest that despite IaC scripts accounting for only 28% of the total number of reviewed files as found in RQ1, the IaC code changes are given equal importance as Non-IaC code. Overall, we observe that while most attributes show statistically significant differences between IaC and Non-IaC code changes, these differences have negligible effect sizes. Our findings indicate that IaC requires at least the same review duration, number of reviewers, number of revisions as Non-IaC code changes, even though IaC constitutes a smaller portion compared to Non-IaC code changes.

To gain a better understanding of the code review process for IaC code changes, we further analyze these attributes according to the three phases of the OpenStack development cycle (development, release candidate, and post-release). In Fig. 8, we show the distribution of IaC and Non-IaC code review attributes across these phases. Three key findings emerged from this analysis.

Finding 4: During the development and release-candidate phases, reviewers tend to exchange more messages when reviewing IaC code changes compared to Non-IaC changes. In Fig. 8, we show that during the development phase, reviewers exchange a higher number of messages while reviewing IaC code changes (with a median of 15 messages) compared to Non-IaC changes (with a median of 13) with a p-value $< 2.2e-16$ and a negligible effect size (0.1). We conjecture that the high number of exchanged messages reflects the effort needed to ensure that the code changes are implemented correctly, and that potential implementation issues are identified and addressed. A similar pattern is observed in the release candidate phase, where the median number of messages exchanged during IaC-related code reviews (median of nine messages) is higher than that for Non-IaC changes (median of 8) with a p-value equal to $3.37e-05$ with a negligible effect size (0.13).

³⁴ <https://review.opendev.org/c/openstack/puppet-neutron/+/382551>

Finding 5: Our analysis reveals that IaC code changes take longer to review during the development phase, but are reviewed faster than Non-IaC code changes during the release-candidate and post-release phases, even when the same number of reviewers is assigned. As depicted in Fig. 8, we identify three main observations.

Development phase During the development phase, we find that IaC code changes take longer to review than Non-IaC code changes, even though more reviewers are assigned on average to IaC code changes. Specifically, IaC code changes take 1.2 times longer to review than Non-IaC code changes. This can be because IaC code changes involve more files (a median of two files) and a larger number of changes, as measured by the churn size (a median of 21 churn), added lines (a median of 13 lines), and deleted lines (a median of three lines). Statistical analysis shows that these differences are statistically significant ($p < 0.05$) with a small to negligible effect size. However, we found in some cases that IaC code changes that were assigned a very high number of reviewers were reviewed faster, *i.e.*, less than the median duration of 92.28 hours during the development phase. For example, the code change ID:611613³⁵ took only 25 hours to be merged but involved 16 reviewers.

It is worth noting that implementing IaC code can often be challenging, as it requires carefully defining complex, interconnected systems, ensuring idempotency, and accurately translating configurations across multiple environments to avoid misconfigurations that could compromise the infrastructure (Guerriero et al. 2019). On the one hand, in some IaC code changes, we found that invited reviewers may lack clarity or consensus on resolving certain issues. For example, the code review ID:316241³⁶ from the *puppet-nova* project took approximately 298 hours (~12.4 days), 13 reviewers, and an amount of exerted 470 code churn to be eventually merged during the development phase of the Newton OpenStack release. The code change updates the API version through a set of configuration parameters that impact four files. Despite the collaborative effort, it underwent 11 revisions, during which reviewers struggled with decisions about which parameters to retain or deprecate and their corresponding values. This struggle is evident in comments such as: *“This v2.2 confuses me, but it was there before so [...]”* and *“This is probably wrong, there is no nova v3.”* The reviewers also highlighted their uncertainty regarding parameter deprecation timing, as expressed in the statement: *“we don’t know when we will remove? I think it’s good to have an idea”*. On the other hand, in some cases, reviewers may require more experienced reviewers to join the review process. This is evident in a comment from a reviewer in code change ID:573657³⁷ from the *puppet-glance* project: *“I’m ok with it, but I want someone from CI team to look at it”*. Furthermore, in the code change ID:26547³⁸, a reviewer requested feedback from an OpenStack project team on 09-01, commenting: *“We need some feedback from glance team.”*, and the glance project team member joined on 11-01. Such statements underscore the importance of involving domain experts to ensure the accuracy and reliability of IaC changes.

Release-candidate phase During the release-candidate phase, our analysis shows that IaC code changes are reviewed and merged faster than Non-IaC code changes, with a 1.2 times difference in review duration. Specifically, IaC code changes have a median review duration of 20.68 hours, while Non-IaC code changes have a median review duration of 23.71 hours. This faster review and merge duration for IaC code changes suggests that reviewers are committed

³⁵ https://review.opendev.org/c/openstack/openstack-ansible-os_neutron/+611613

³⁶ <https://review.opendev.org/c/openstack/puppet-nova/+316241>

³⁷ <https://review.opendev.org/c/openstack/tripleo-ci/+573657>

³⁸ <https://review.opendev.org/c/openstack/puppet-glance/+265470>

to quickly addressing IaC bugs and resolving them before the new OpenStack release is deployed in production environments. As explained in Section 2.1.3, the IaC code changes merged during the release-candidates and post-releases are bug-focused. For instance, the code change ID:430519³⁹ of the project *openstack-ansible-os_keystone* took place during the release-candidate phase to fix a bug, which required a churn amount of 78 and was merged in less than nine hours. While IaC code changes are reviewed and merged more quickly during the release candidate phase, they tend to involve more files (a median of two files) and require more changes (a median churn of 12, median added lines of nine, and median deleted lines of two) than Non-IaC code changes (median of one file, churn of two, added lines of one, and deleted lines of one). These differences are statistically significant ($p < 0.05$) with a small effect size, which suggests that fixing IaC bugs during the release candidate phase may be more difficult and time-consuming as they impact a larger number of files. However, the importance of quickly addressing these bugs may outweigh the additional effort required. These findings also support the idea that IaC bugs might be given a higher priority during the later stages of the release cycle or that they are less complex and easier to review, which motivates a survey with IaC experts.

Post-release phase During the post-release phase, our analysis shows that IaC code changes, which are bug-focused (OpenStack 2022, 2016a), are reviewed and fixed quicker than Non-IaC bugs. IaC code changes takes a median of 45 hours to be merged during the post-release, compared to 54 hours for Non-IaC code changes. However, we notice that the review duration for IaC code changes is longer during the post-release phase than during the release-candidate phase. This might be explicable by the prioritization of identified bugs, as bugs of high-priority level must be fixed before the release date, and non-critical bugs are left to be fixed post-release (Teixeira and Karsten 2019). Furthermore, the tight time frame of a release candidate phase could be a factor that urges developers and reviewers to provide bug fixes promptly. For example, the code change ID:235356⁴⁰ of the *puppet-neutron* project involved 10 reviewers to be finally merged across the span of 54 days during the post-release phase. Furthermore, our analysis shows that fixing IaC bugs during the post-release phase tends to involve more files (median of two), require more churn (median of 17), and more added lines (median of 12), compared to Non-IaC code changes (median of one file, churn of 10, and added lines of seven) with a statistical significance of a p-value $< 2.2e-16$ and a small effect size. This finding suggests that developers need to be particularly careful when implementing changes requested by reviewers during the IaC code review process, given the larger number of files involved. Future research may prompt us to examine the nature of the IaC bugs identified in more detail.

We present in Table 6 the Multiple Regression Analysis (MRA) results for both IaC and Non-IaC code changes. For each variable, we report its coefficients (Coeff), the corresponding percentage of sum of squares (% Sum Sq) as a measure of variance explained by the variable, and the standard error of the regression (Error), which reflects the average distance between the observed values and the regression line. Statistical significance is indicated by asterisks. We consider a coefficient to be important if it is statistically significant ($p < 0.05$).

The regression results indicate that the confounding variables (size, age, and cycle) are statistically significant for 86% of the attributes (p-value < 0.05). However, their impact on the variance of the attributes is considered small with a low percentage of sum squares for all confounding variables, except for the size reflecting 10% of the variation in the number of

³⁹ https://review.opendev.org/c/openstack/openstack-ansible-os_keystone/+/430519

⁴⁰ <https://review.opendev.org/c/openstack/puppet-neutron/+/235356>

Table 6 Multiple Regression Analysis results for the confounding variables {size, age, cycle} impact on the code change review attributes for IaC and Non-IaC code changes

| Attributes | Variables | IaC | | | | Non-IaC | | | |
|------------------------|------------|---------|-------|-----|-----------|---------|-------|-----|-----------|
| | | Coeff | Error | p | % Sum Sq. | Coeff | Error | p | % Sum Sq. |
| Duration | Intercept | 327.16 | 17.07 | *** | | 217.21 | 3.80 | *** | |
| | log(Size) | -0.95 | 2.95 | | 0.001 | 0.89 | 0.48 | . | 0.001 |
| | log(Age) | 3.91 | 1.95 | * | 0.04 | 13.97 | 0.44 | *** | 0.42 |
| | log(Cycle) | -145.29 | 14.06 | *** | 1.25 | -69.36 | 3.59 | *** | 0.15 |
| | R^2 | 0.012 | | | | 0.005 | | | |
| No. of revisions | Intercept | 4.61 | 0.09 | *** | | 4.64 | 0.02 | *** | |
| | log(Size) | -0.15 | 0.01 | *** | 0.88 | 0.02 | 0.002 | *** | 0.02 |
| | log(Age) | 0.04 | 0.01 | *** | 0.15 | -0.08 | 0.002 | *** | 0.46 |
| | log(Cycle) | -1.89 | 0.08 | *** | 5.94 | -1.91 | 0.02 | *** | 3.58 |
| | R^2 | 0.07 | | | | 0.04 | | | |
| No. of reviewers | Intercept | 5.35 | 0.09 | *** | | 4.77 | 0.02 | *** | |
| | log(Size) | 0.16 | 0.01 | *** | 1.08 | 0.16 | 0.002 | *** | 1.33 |
| | log(Age) | -0.004 | 0.01 | | 0.002 | -0.01 | 0.002 | *** | 0.02 |
| | log(Cycle) | -1.21 | 0.07 | *** | 2.75 | -0.61 | 0.02 | *** | 0.34 |
| | R^2 | 0.03 | | | | 0.01 | | | |
| Added lines | Intercept | 10.66 | 0.71 | *** | | 7.76 | 0.13 | *** | |
| | log(Size) | 2.40 | 0.12 | *** | 3.92 | 2.19 | 0.01 | *** | 6 |
| | log(Age) | -2.40 | 0.08 | *** | 8.91 | -1.94 | 0.01 | *** | 5.66 |
| | log(Cycle) | 2.34 | 0.58 | *** | 0.16 | 1.70 | 0.12 | *** | 0.06 |
| | R^2 | 0.10 | | | | 0.09 | | | |
| Deleted lines | Intercept | 3.68 | 0.36 | *** | | 2.51 | 0.07 | *** | |
| | log(Size) | -0.22 | 0.06 | *** | 0.14 | -0.05 | 0.009 | *** | 0.01 |
| | log(Age) | 0.70 | 0.04 | *** | 3.21 | 0.43 | 0.008 | *** | 1.14 |
| | log(Cycle) | -2.85 | 0.30 | *** | 1.02 | -0.81 | 0.06 | *** | 0.06 |
| | R^2 | 0.04 | | | | 0.01 | | | |
| No. of files | Intercept | 19.97 | 0.57 | *** | | 28.64 | 0.22 | *** | |
| | log(Size) | -3.13 | 0.09 | *** | 10.18 | -1.70 | 0.02 | *** | 1.43 |
| | log(Age) | 1.14 | 0.06 | *** | 3.11 | -1.12 | 0.02 | *** | 0.74 |
| | log(Cycle) | -5.88 | 0.46 | *** | 1.59 | -3.1 | 0.21 | *** | 0.08 |
| | R^2 | 0.12 | | | | 0.03 | | | |
| No. of inline comments | Intercept | 2.95 | 0.14 | *** | | 3.35 | 0.03 | *** | |
| | log(Size) | 0.01 | 0.02 | | 0.006 | 0.11 | 0.004 | *** | 0.32 |
| | log(Age) | -0.03 | 0.01 | * | 0.05 | -0.12 | 0.003 | *** | 0.47 |
| | log(Cycle) | -1.95 | 0.11 | *** | 3.17 | -2.09 | 0.03 | *** | 1.91 |
| | R^2 | 0.03 | | | | 0.02 | | | |

Table 6 continued

| Attributes | Variables | IaC | | | | Non-IaC | | | |
|--------------------|------------|--------------|--------------|----------|------------------|--------------|--------------|----------|------------------|
| | | <i>Coeff</i> | <i>Error</i> | <i>p</i> | <i>% Sum Sq.</i> | <i>Coeff</i> | <i>Error</i> | <i>p</i> | <i>% Sum Sq.</i> |
| No. of messages | Intercept | 20.39 | 0.39 | *** | | 19.15 | 0.08 | *** | |
| | log(Size) | 0.27 | 0.06 | *** | 0.18 | 0.42 | 0.01 | *** | 0.67 |
| | log(Age) | -0.03 | 0.04 | | 0.006 | -0.17 | 0.009 | *** | 0.12 |
| | log(Cycle) | -6.32 | 0.32 | *** | 4.21 | -5.58 | 0.07 | *** | 2.07 |
| | R^2 | 0.02 | | | | 0.02 | | | |
| Description length | Intercept | 33.48 | 0.50 | *** | | 32.68 | 0.10 | *** | |
| | log(Size) | 0.49 | 0.08 | *** | 0.38 | 0.86 | 0.01 | *** | 1.81 |
| | log(Age) | 0.11 | 0.05 | * | 0.05 | -0.13 | 0.01 | *** | 0.05 |
| | log(Cycle) | 3.79 | 0.41 | *** | 0.98 | 3.33 | 0.09 | *** | 0.48 |
| | R^2 | 0.01 | | | | 0.02 | | | |

***: $p < 0.001$, **: $p < 0.01$, *: $p < 0.05$, ‘.’: $p < 0.1$, ‘ ’: $p > 0.1$

files. Furthermore, the significance of the confounding variables in explaining the variation in the attributes remains limited, as reflected by the low R^2 values. Particularly, we observe that the cycle has higher coefficient than age and size in all attributes except for the Added lines. For example, the cycle has a high negative coefficient of 145.29 for the duration of IaC code changes and a negative coefficient of 69.36 for the Non-IaC code changes. That is, for one unit variation in cycle, the duration of IaC code changes tends to decrease by 145.29 hours and by 69.36 hours for the Non-IaC code changes. However, the cycle only explains less than 1% of the variance in the duration. Still, we conducted our analysis based on the cycle phases (development, release-candidate, and post-release). The results indicate that while statistical significance is found in many cases, the variables explain a small proportion of the variation in code change attributes for IaC and Non-IaC code changes. These findings suggest that the confounding variables are not impacting our initial conclusions and our results are valid. i.e., the difference between the code review attributes for IaC and Non-IaC code changes is due to the type of files (IaC or Non-IaC) and not the size, age, or cycle.

4.3 RQ3: What is the Contribution of Reviewers Working on IaC Code Changes?

Motivation Despite that IaC code changes represent only 10% of the studied code changes, the median number of reviewers assigned to IaC code changes is no different from those assigned to Non-IaC code changes, and also the review process for IaC code changes takes the same duration as for Non-IaC code changes. Therefore, we want to extract the contribution of reviewers involved in IaC code changes. Prior studies (Peruma et al. 2019; AlOmar et al. 2021) have shown that more experienced developers are often more involved in specific code changes, e.g., changes related to code refactoring. In this research question, we aim to investigate if reviewers with more experience, as measured by their contributions, are more likely to review IaC code changes.

Approach To examine the reviewers’ contribution to the OpenStack review process, we utilize the number of reviewed and authored code changes as a proxy for their contribution. Similarly to previous studies (Peruma et al. 2019; AlOmar et al. 2021), we compute the Reviewer Contribution Score (RCS) for each reviewer in our dataset (cf. Section 3 Step 3.1).

Then, we further inspect whether the reviewers involved in IaC code changes are among the most active; i.e., contribute more to the review and development process of OpenStack.

Results As shown in Fig. 9, we distinguish between two types of contributions, (1) reviewers who review and author only Non-IaC code changes (depicted as “Non-IaC”) and (2) reviewers who review at least one IaC code change (depicted as “IaC”). We found that our dataset includes a significant number of reviewers who contributed to Non-IaC code changes. However, these reviewers do not have the highest RCS scores.

Finding 6: We observe that reviewers who review and author IaC code changes are more contributing to the review process of OpenStack than Non-IaC reviewers. A clear observation from Fig. 9 shows that the two types (IaC and Non-IaC) of reviewers’ contributions follow the same pattern throughout the 12 OpenStack releases. The density plot distinctly illustrates a left-skewed to evenly distributed data pattern for IaC reviewers’ RCS in contrast to the right-skewed distribution observed for Non-IaC reviewers’ RCS. This suggests that IaC reviewers made more contributions to the OpenStack review process compared to Non-IaC reviewers. Although there was some overlap in the density plot, the majority of reviewers who reviewed Non-IaC code changes had lower RCS scores, while IaC reviewers had higher RCS scores. Statistically, IaC reviewers had the highest RCS scores, with maximum scores ranging from 0.07 to 0.22. Although Non-IaC reviewers were more prevalent in the dataset, they had the lowest RCS scores, with a maximum range from 0.03 to 0.12.

Similarly to AlOmar et al. (2021), we perform a non-parametric Mann-Whitney Wilcoxon rank-sum test on the RCS values for IaC and Non-IaC reviewers for the 12 OpenStack releases. We obtained statistically significant p-values (< 0.001) with a large effect size between the two groups of reviewers. Therefore, this suggests that Non-IaC reviewers are less contributing

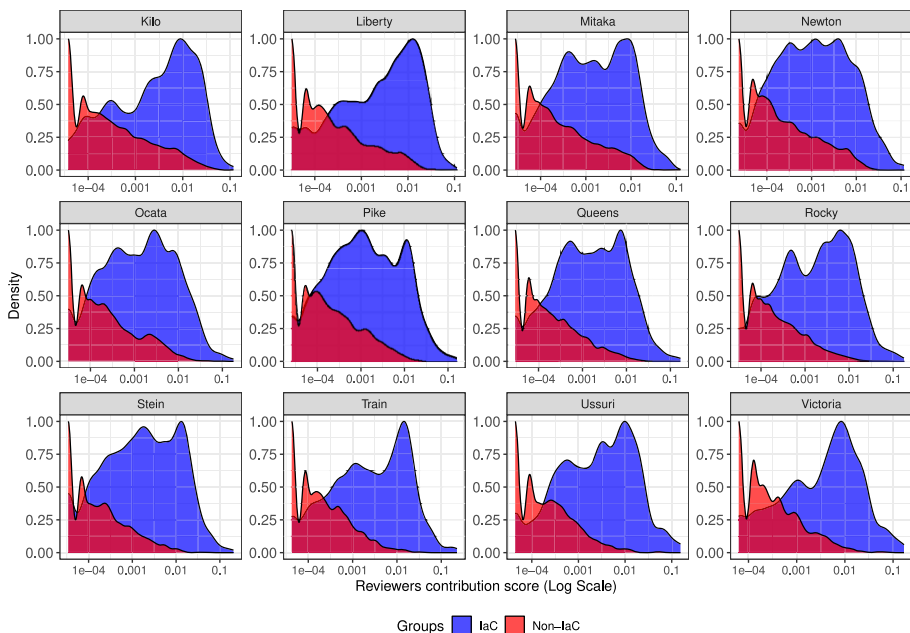


Fig. 9 Contribution score of reviewers working on *IaC* and *Non-IaC* code changes of OpenStack releases

to the review process of OpenStack development than those who review both IaC and Non-IaC code changes (IaC reviewers). These results may indicate that IaC reviewers are more engaged in all code changes that involve an interconnectedness between IaC and Non-IaC code changes. IaC reviewers likely possess specialized knowledge and expertise related to infrastructure configurations, which enables them to make more substantial contributions to both IaC and Non-IaC code changes. Our analysis suggests that a dedicated group of developers is primarily responsible for reviewing and developing IaC code changes. However, these developers also participate in reviewing Non-IaC code changes, indicating a collaborative effort between IaC and Non-IaC teams. These findings lead us to investigate whether the dedicated IaC reviewers are among the active reviewers in OpenStack.

Finding 7: Developers that are mostly contributing to the review process of IaC code changes are the most involved in the development of IaC code changes, as presented in Fig. 10. First, we found that the top-5% reviewers are reviewing 44% to 75% of the total number of IaC code changes than the rest of the reviewers. Thus, we advocate that dedicated/experienced reviewers be assigned to heavily review IaC-related code changes. Figure 10a shows the distribution of reviewed IaC code changes for both top-5% and remaining reviewers (i.e., 95%). The first notable observation is the volume of IaC code changes reviewed by both groups of reviewers. The top-5% reviewers have a higher contribution to IaC code changes review process compared to the rest of the reviewers. On average, the top-5% reviewers reviewed 12 to 88 IaC code changes in the 12 OpenStack releases (the maximum ranges from 439 to 1,497), whereas the rest of the reviewers average around one to two IaC code changes (the maximum ranges from 57 to 186), despite being more numerous in the dataset. A non-parametric Wilcoxon rank-sum test was conducted on the number of IaC code changes reviewed by both the top-5% and the rest of the reviewers and showed a statistically significant p-value ($<2.2e-16$) between the two groups of reviewers in the 12 OpenStack releases, with a medium effect size in the Kilo release and a large effect size in all other releases.

Furthermore, we also found that the top-5% of reviewers are authoring more IaC code changes than the remaining 95% of reviewers, as presented in Fig. 10b. On average, the top-5% reviewers authored three to 12 code changes in the 12 OpenStack releases (the maximum ranges from 95 to 339), whereas the rest of the reviewers average around one to two code changes (the maximum ranges from one to 124), despite being more numerous in the dataset. A non-parametric Wilcoxon rank-sum test was conducted on the number of IaC code changes

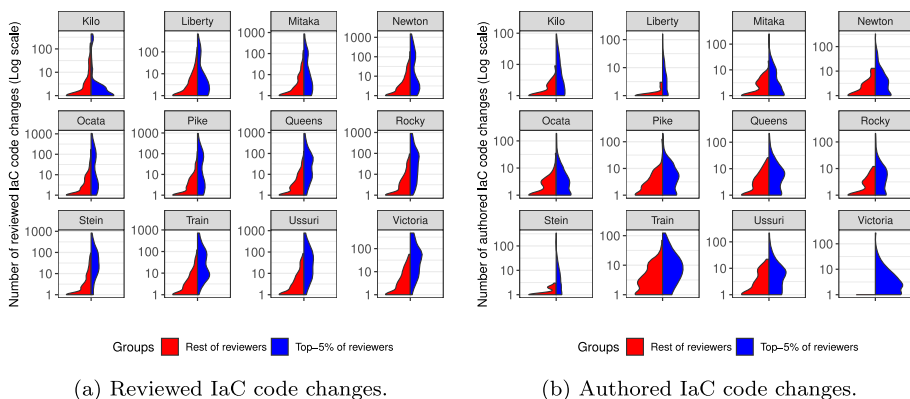


Fig. 10 The distribution of IaC code changes (a) reviewed and (b) authored by the Top-5% of reviewers compared to the rest 95% of reviewers

authored by both the top-5% and the rest of the reviewers and showed a statistically significant p-value (<0.001) between the two groups of reviewers in the 12 OpenStack releases, with a medium to large effect sizes.

4.4 RQ4: What Criteria are Considered during the Code Review to Merge IaC Code Changes?

Motivation The second research question (RQ2) revealed that during the development and release-candidate phases, there is a higher number of messages exchanged by reviewers when reviewing IaC code changes compared to Non-IaC changes. Furthermore, we found in RQ3 that the top-5% of these reviewers participate in 44% to 75% of IaC code changes. Therefore, we sought to determine the criteria that reviewers discuss and consider when merging an IaC code change during the three phases and present them as a checklist. Similarly to Kumara et al. (2021) who established a checklist for the bad and good practices of IaC development, we explore this practice from a code review perspective. This checklist can assist developers in identifying IaC-related issues they might not have been aware of, and may also help improve the quality of IaC code changes.

Approach To get a more qualitative sense, we manually analyze the discussion and inline comments of reviewers in IaC-related code changes. Our objective is to understand the most common issues and quality standards that are addressed during the review process of IaC code changes. This will provide us with a better qualitative understanding of what needs to be considered when coding IaC. Furthermore, we surveyed nine of the most active OpenStack reviewers to validate our checklist.

Results Our checklist, as shown in Fig. 11, is composed of two layers: the top layer contains seven generic topics that group issues with similar context, whereas the lower layer encompasses 38 issues that provide a fine-grained explanation of the generic topics discussed during the IaC code review.

Finding 8: Our analysis identifies seven main topics related to IaC that reviewers discuss, encompassing a total of 38 specific issues. As shown in Fig. 12, the checklist consists of seven generic topics: (1) Logic, (2) Best-practices, (3) Documentation, (4) Compatibility, (5) Dependencies, (6) Test, and (7) Security. According to the survey findings, out of the total 324 ratings given by the developers, we only have received five negative ratings of “*I disagree*” (1) and “*I somewhat disagree*” (4), resulting in a 98% acceptance rate of the checklist items. Four participants gave the rating “*disagree*” (1) and “*I somewhat disagree*” (3) to the best-practice related item “*Always be aware of new IRC conventions and upgrades.*” for not being clear. Therefore, we rephrase our item to “*Always be aware of OpenStack Internet Relay Chat (IRC) conventions.*” For the same reason, another participant gave “*I somewhat disagree*” (1) rating to the documentation related item “*Add a bug report to document bug fixes*” for the same reason. Thus, we rephrase our item to “*Add a bug report to document new detected bugs.*” Additionally, two reviewers suggested including two new items under the topics of “*Security*” and “*Logic*”. We provide a detailed description of each topic and further clarify our checklist items. These topics provide a high-level context for 38 specific issues, which are detailed in Table 7.

(T1) Logic This topic encompasses logic-related issues, such as incorrect values assigned to configuration options or code placed in the wrong location. It is the first most discussed topic by reviewers when reviewing IaC code changes, accounting for 33% of discussions.

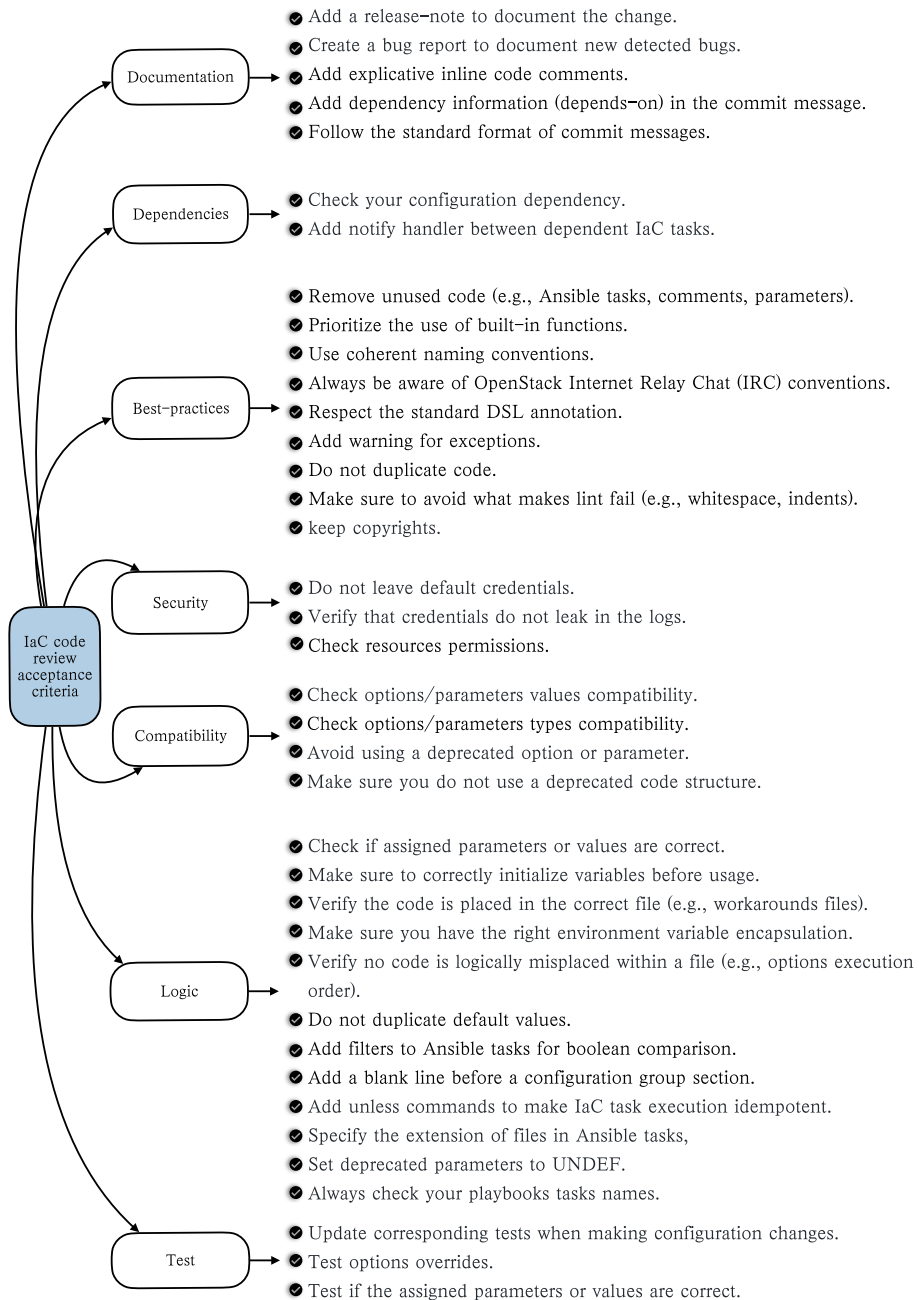


Fig. 11 The 38 identified IaC code review acceptance criteria using thematic analysis

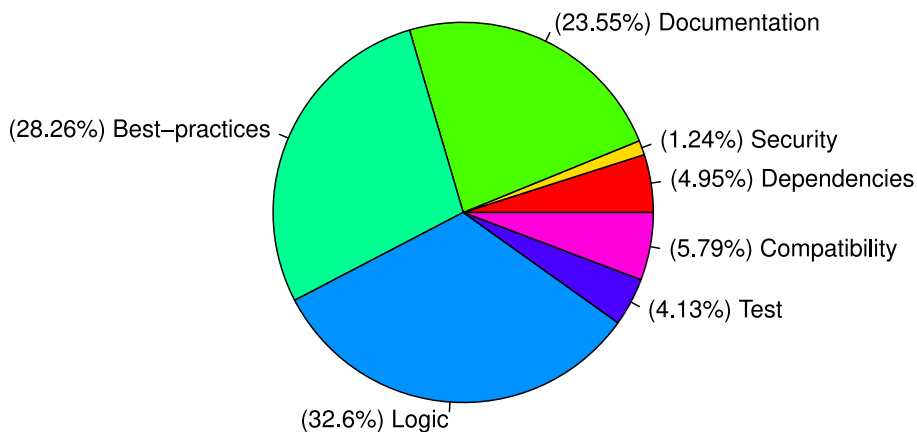


Fig. 12 The distribution of the seven identified topics in IaC code changes

Table 7 A taxonomy of 38 IaC code changes review checklist in Code Review

| Topic | Checklist item | Examples (from IaC review discussions) |
|--------------------|--|--|
| (T1) Documentation | Add a release-note to document the change. | <i>"The code looks good, but could you please add a release note?"</i> (item example 2021a) |
| | Add a bug report to document new detected bugs. | <i>"Please create a bug report for this issue and link it to the change."</i> (item example 2021b) |
| | Add explicative inline code comments. | <i>"Can you document what each one means in this context please?"</i> (item example 2021c) |
| | Add dependency information (dependencies) in the commit message. | <i>"Needs Depends-On: in commit message for puppet-tripleo dependency which adds this."</i> (item example 2021d) |
| | Follow the standard format of commit messages. | <i>"Please follow proper formatting for commit messages: Title of commit < less than 50 characters <CR> Description of commit (wrapped at 71 characters) may be multiline Closes-Bug: #X or TrivialFix or Implements."</i> (item example 2021e) |
| (T2) Logic | Check if assigned parameters or values are correct. | <i>"Should default to false."</i> (item example 2021f) |
| | Make sure to correctly initialize variables before usage. | <i>"We spotted it misses hiera ('rabbit_ipvs')." (item example 2021g)</i> |
| | Verify the code is placed in the correct file (e.g., workarounds files). | <i>"I just realized this is the wrong file, it should be spec/classes/octavia_health_manager_spec.rb not octavia_worker_spec.rb."</i> (item example 2021h) |
| | Make sure you have the right environment variable encapsulation. | <i>"Should we add 'export'. It might be quite useful to have it between other scripts."</i> (item example 2021i) |

Table 7 continued

| Topic | Checklist item | Examples (from IaC review discussions) |
|------------------------|--|---|
| (T3) Best-practices | Verify no code is logically misplaced within a file (e.g., options execution order). | <i>"You want this in line 29 instead."</i> (item example 2021j) |
| | Do not duplicate default values. | <i>"That's the default, remove it."</i> (item example 2021k) |
| | Add filters to Ansible tasks for boolean comparison. | <i>"It's common in ansible to set flags to "no" but without a bool filter this will be evaluated as true."</i> (item example 2021l) |
| | Add a blank line before a configuration group section. | <i>"Should add a blank line before this section."</i> (item example 2021m) |
| | Add unless commands to make IaC task execution idempotent. | <i>"This is a task that basically is being used by other puppet resources to ensure that rabbitmq is ready. If the unless is true it's "skipped" by puppet but still put in a specific place for ordering which is what we need. If anything was subscribed to this they would not get notifications, thus it would be properly idempotent."</i> (item example 2021n) |
| | Specify the extension of files in Ansible tasks. | <i>"You should add the .yaml extension (zuul developers said that they will remove the support for automatically adding the extension)."</i> (item example 2021o) |
| | Set deprecated parameters to UNDEF. | <i>"Deprecated parameters are usually set to undef by default, so we know they do nothing."</i> (item example 2021p) |
| | Always check your playbooks tasks names. | |
| | Remove unused code (e.g., Ansible tasks, comments, parameters). | <i>"Is this var needed?"</i> (item example 2021q) |
| | Prioritize the use of DSL built-in functions. | <i>"Please use empty function from stdlib."</i> (item example 2021r) |
| | Use coherent naming conventions. | <i>"All the other python-tempestconf variables have a "tempest_" prefix except these two."</i> (item example 2021s) |
| | Always be aware of OpenStack Internet Relay Chat (IRC) conventions. | <i>"This condition ansible_architecture == "ppc64le" for each task inside this file is not required when ansible version is >= 2.0. Because include statement for this file in the caller nova_compute_kvm.yml already has this condition."</i> (item example 2021t) |
| | Respect the standard DSL annotation. | <i>"Could you use yaml notation here?"</i> (item example 2021u) |
| | Add warning for exceptions. | <i>"Add a notice/warning here."</i> (item example 2021v) |
| | Do not duplicate code. | <i>"I find it sad that we would have to duplicate this logic everywhere."</i> (item example 2021w) |

Table 7 continued

| Topic | Checklist item | Examples (from IaC review discussions) |
|--------------------|--|---|
| (T4) Compatibility | Make sure to avoid what makes lint fail (e.g., whitespace, indents). | <i>"This trailing whitespace is making the puppet lint job fail."</i> (item example 2021x) |
| | keep copyrights. | <i>"The previous copyright should not be removed."</i> (item example 2021y) |
| | Check options/parameters values compatibility. | <i>"Fair enough - Seems to be consistent for other services. As mentioned in IRC though, considering that this is a microservice, we should change the default in novajoin to 127.0.0.1 by default."</i> (item example 2021z) |
| | Check options/parameters types compatibility. | <i>"So one of those minor backwards compatibility things, \$provider_mappings = false will cause this to error because empty only works with array, hashes, strings and integers."</i> (item example 2021aa) |
| (T5) Dependencies | Avoid using a deprecated option or parameter. | <i>"Usage of cinder::volume::dellsc_iscsi is deprecated, please use cinder::backend::dellsc_iscsi instead."</i> (item example 2021ab) |
| | Make sure you do not use a deprecated code structure. | <i>"Don't use bare variable in condition, this is deprecated since Ansible 2.8."</i> (item example 2021ac) |
| | Check your configuration dependency. | <i>"How do we ensure that this is executed after all repos have been configured?"</i> (item example 2021ac) |
| | Add notify handler between dependent IaC tasks. | <i>"Notify the 'Reload systemd daemon' instead of manually doing it."</i> (item example 2021ae) |
| (T6) Test | Update corresponding tests when making configuration changes. | <i>"Could you please cover this case in tests?"</i> (item example 2021af) |
| | Test options overrides. | <i>"I am not sure is it wise idea to override a fact in ansible, can you make it a single expression?"</i> (item example 2021ag) |
| | Test if the assigned parameters or values are correct. | |
| (T7) Security | Do not leave default credentials. | <i>"Leaving default credentials is a terrible idea for security. we should enforce the parameters by no leaving default values."</i> (item example 2021ah) |
| | Verify that credentials do not leak in the logs. | <i>"I think we should add a logoutput to false, so if registration fails we don't leak the credentials in the logs."</i> (item example 2021i) |
| | Check resources permissions. | |

In 43% of the cases, reviewers identified incorrect values being assigned to parameters or options. This is a particularly important issue as incorrect values for these options can cause significant errors in IaC, as pointed out by Sayagh et al. (2017). In 26% of the cases, reviewers emphasized the importance of assigning deprecated options to the UNDEF value without any deprecation period to ensure that these options do not affect the system. For example, in the code change ID:491309, one of the reviewers commented: *"should we just switch this to*

undef now since it has no effect otherwise we'll always generate that warning" (OpenDev 2017d). Additionally, 13% of the issues involved the need to add file extensions to Ansible tasks, as they are no longer added automatically. We support this by the comment of one of the reviewers in ID:509242, saying: *"You should add the .yaml extension (zuul developers said that they will remove the support for automatically adding the extension)"* (OpenDev 2017c). In 9% of the cases, reviewers pointed out a code that was misplaced within a file. In 7% of the cases, they identified a code that was placed in the wrong file, such as placing the workaround code in external workaround files or configuration section.⁴¹ For example, we find in the code change ID:626637 that a reviewer spotted misplaced code and explicitly commented: *"I just realized this is the wrong file, it should be spec/classes/octavia_health_manager_spec.rb not octavia_worker_spec.rb"* (OpenDev 2019b). Finally, in a small percentage (1%) of the cases, reviewers identified issues related to the idempotency of the IaC code, which refers to the ability of the code to produce the same output in different environments. Kumara et al. (2021) also discouraged any IaC coding practices that defy the idempotency of IaC scripts. One of the surveyed reviewers has also provided an additional item that needs to be checked related to naming the Ansible Playbooks tasks names. We have added the new recommended point to our items under *"Always check your playbooks tasks names."* Another participant has stated: *"I found this category of logic checklist quite comprehensive. This is important not only for newcomers in the project but also for both code change owners and reviewers."*, which empowers the usefulness of our logic-related items.

(T2) Best-practices We found that 28% of the reviewers' comments focused on best practices for IaC coding. Out of these comments, 29% were concerned with removing unused or dead code, such as removing unnecessary IaC tasks, options, parameters, or code comments that are no longer needed. For example, in the code change ID:307419, a reviewer requested to remove a variable that is not needed, as they commented: *"is this var needed?"* (OpenDev 2016a). In the second most common best practices-related issue, accounting for 28%, the reviewers requested that developers include copyrights in the code files. One of the surveyed developers indicated that *"Copyrights are red line for any code review"*. Furthermore, 16% covered issues such as avoiding duplication of default values for configuration options and adhering to community conventions. Additionally, 13% was related to the proper use of annotations in the IaC tool's language (DSL), which is in line with the identified best practice in Kumara et al. (2021) related to *"Make code style and formatting consistent"*. Many comments requested that developers follow standard DSL annotations, such as YAML notation. In addition, 8% of the comments addressed the importance of using conventional names for options and parameters, while in the remaining 6% of the cases, reviewers identified issues related to the Lint tool, which checks for syntax errors and other cosmetic problems such as line length, trailing spaces, and indentation. Developers should be aware of these types of errors and try to avoid them to avoid wasting reviewers' time.

(T3) Documentation Documenting code changes is crucial for keeping track of the current state of the infrastructure. We observed that 23.55% of the most common reviewer concerns were related to documentation. This highlights the importance placed by reviewers on documenting IaC code changes. However, it also reveals the gap in developers' understanding of the importance of documentation in IaC. In 65% of the comments, reviewers requested updates to the documentation (commit message or release notes) to better describe the purpose of the IaC code change or to provide more detailed information. The second most common documentation-related issue is the need to add a release note to document the change that

⁴¹ <https://docs.openstack.org/nova/pike/admin/root-wrap-reference.html>

occurred in 19.3% of the cases. For instance, in the IaC code change ID:528250, a reviewer commented: *“The code looks good, but could you please add a release note?”* (OpenDev 2017b). Another 7% of the comments focused on the lack of explanatory code comments, where reviewers ask for more documentation in the IaC code files. For example, in the code change ID:509186, a reviewer asked for a more explanatory code comment: *“Can you document what each one means in this context please?”* (OpenDev 2017a). Overall, reviewers encourage adopting the best documentation practices into the development process, such as through code comments or separate documentation files. Our findings align with the IaC best practices list established by Kumara et al. (2021) about documentation.

(T4) Compatibility In this topic, we identified 6% of the issues related to compatibility that occur when different resources, such as options, IaC tasks, or parameters do not work together as intended, either because of incompatible values (i.e., the values of the options do not work well together or cause conflicts) or types (i.e., the options do not possess the right type) or because deprecated code is being used. For example, in 64.29% of the cases, reviewers identified the use of deprecated code structures from older versions of IaC DSL. For instance, one reviewer identified the use of a deprecated code structure from an older version of the Ansible IaC tool, stating: *“Don’t use bare variable in condition, this is deprecated since Ansible 2.8.”* (OpenDev 2019a). In 21% and 7% of the cases, reviewers pointed out issues related to the use of deprecated options or parameters that are no longer functional in the system and the use of incompatible parameter types that could cause conflicts. These findings highlight the importance of ensuring compatibility in IaC code and avoiding the use of deprecated code structures.

(T5) Dependencies In 5% of the comments, reviewers focused on issues related to dependencies within the system. The majority of these comments (83%) concerned configuration dependencies. In particular, reviewers discussed the order in which system resources, such as packages, should be configured. For example, one reviewer asked: *“How do we ensure that this is executed after all repo have been configured?”* (OpenDev 2018). Incorrectly managing dependencies can lead to broken dependencies and unexpected system behavior, so it is important to clearly document the configuration workflow. The remaining 17% of comments in this topic focused on notifications for service changes and notifications for dependent Ansible tasks with their execution states. For example, one of the reviewers in the code change ID:391532 commented: *“Notify the ‘Reload systemd daemon’(task) instead of manually doing it”* (OpenDev 2016b). These findings underscore the importance of carefully managing dependencies in IaC code to avoid bugs and ensure smooth system operation.

(T6) Test Testing is crucial to ensure that code changes do not negatively impact the system’s functionality. Similar to Kumara et al. (2021), we also found that developers usually test as they code, as only a tiny 4% of the code changes reviewed included testing issues. In 80% of these cases, reviewers requested updates to the tests to include different scenarios. This suggests that the tests may not be sufficient to fully test IaC code, as they may not cover all possible scenarios. As one reviewer in the code change, ID:750656 commented: *“We should really test whatever it is that users are expected to do.”* (OpenDev 2020). It is important for developers to regularly update and validate their tests to cover all new changes to the IaC code, as it may be interleaved with other production tasks. In the remaining 20% of the cases, reviewers requested specific types of testing, such as testing of overridden options to

Table 8 The seven topics distribution in the three OpenStack cycle phases: development, release-candidate, and post-release

| Topics | Development | Release-candidate | Post-release |
|---------------------|-------------|-------------------|--------------|
| (T1) Logic | 56% | 2% | 42% |
| (T2) Best-practices | 56% | 4% | 40% |
| (T3) Documentation | 56% | 2% | 42% |
| (T4) Compatibility | 64% | 7% | 29% |
| (T5) Dependencies | 25% | 8% | 67% |
| (T6) Test | 60% | 10% | 30% |
| (T7) Security | 67% | 0% | 33% |

ensure that the expected values are being used. To prevent bugs and avoid breaking the code, developers should ensure that their tests cover all emerging changes in their IaC code.

(T7) Security Ensuring the security of a software system is critical, particularly in the context of IaC where infrastructure can be vulnerable to threats such as data breaches. While security issues were the least common topic identified, representing only 1% of the comments, we added them to our checklist, as security best practices are of paramount importance in IaC and can lead to severe errors as pointed out by Rahman et al. (2019, 2021). We found 67% of these issues involved concerns about leaving default credentials in the code or leaking their values in logs. For example, in the code change ID:404892, one of the reviewers commented: *“leaving default credentials is a terrible idea for security. we should enforce the parameters by no leaving default values.”* (OpenDev 2016c). We also mention that Kumara et al. (2021) encourage isolating secrets (sensitive information) from code. Developers need to prioritize security in the development process to protect against potential threats. Furthermore, a surveyed developer has recommended a new item related to resources permission to prevent unauthorized components from using a resource if they were mistakenly been given wrong permissions.⁴² We label this new item as *“Check resources permissions”*.

Finding 9: We observe that six topics (Logic, Best-practices, Documentation, Compatibility, Dependencies, and Test) are present in the three phases of OpenStack release cycle except for the Security. In Table 8, we present the percentages of issues related to each topic that are raised during each phase of the OpenStack release cycle. We observe that reviewers consistently identify issues across all three phases, highlighting a need for improved developer communication throughout the release cycle. Our checklist proves useful in preventing the recurrence of issues identified in a previous phase. Notably, six topics, constituting over 51% of all discussed issues, are more prevalent in the development phase. The top three topics mostly discussed in this phase, as opposed to the release-candidate and post-release phases, are *Security* (T7), *Compatibility* (T4), and *Test* (T6). Although these topics may not be extensively discussed overall, they receive higher attention in the initial phase of OpenStack release development due to their critical role in testing system security and compatibility.

Particularly, during the development phase, compatibility is discussed extensively, accounting for 64% of the reviews, because ensuring that scripts work seamlessly across various environments is crucial. Early identification and resolution of compatibility issues prevent deployment failures later in the release cycle. In the release-candidate phase, only

⁴² https://docs.ansible.com/ansible/latest/collections/ansible/builtin/file_module.html

7% of the reviews focus on compatibility, as most issues should have been identified and resolved by then. However, in the post-release phase, discussions on compatibility increase to 29% as real-world usage may uncover unforeseen compatibility problems that need to be addressed to maintain the release's reliability and user satisfaction. The same goes for testing, as it is a critical focus during the development phase, with 60% of the reviews addressing this topic, as it is essential to identify and fix issues early to ensure the IaC scripts function correctly. In the release-candidate phase, testing discussions drop to 10% because the primary focus shifts to final verification and stabilizing the release. In the post-release phase, testing becomes relevant again, accounting for 30% of the reviews, as ongoing maintenance and updates necessitate continuous testing to address any new issues that arise from real-world usage. Similarly, security related issues are a major concern during the development phase, with 67% of the reviews discussing it. Identifying and mitigating potential vulnerabilities early on is crucial to prevent severe issues later. Interestingly, security is not discussed (0%) in the release-candidate phase, possibly because the focus at this stage is on finalizing the product and addressing critical bugs rather than introducing new security measures. In the post-release phase, security discussions increase to 33% as ongoing monitoring and real-world usage may reveal new vulnerabilities that need to be addressed promptly. These insights encourage developing a survey with IaC developers to understand their prior focus on IaC during each phase and why.

Additionally, we observe that issues related to *Dependencies* (T5) are most frequently brought up in the post-release phase, primarily centered on bug fixing. To comprehend why dependencies are a common topic during this bug-focused phase, we examined bug reports submitted to the bug tracking system.⁴³ These reports were specifically associated with code changes where dependency issues were identified in our manually analyzed representative sample of IaC code changes. We found five code changes where dependency issues occurred, and we accessed their bug reports, by clicking on the “Closes-Bug” link in the commit message of each code change. Upon manually analyzing the five related bug reports (bug description and exchanged messages), we identified two main types of dependency-related issues. The first type pertains to issues with package dependencies. For example, in the code change ID:298679, the deployment of the module *puppet-horizon* should be installing the *python-memcache* package as well, which was not the case (Launchpad 2016a), as one of the developers indicated: “*it likely sounds a dependency issue in horizon packaging [...]*”. The second type involves conflicts with configuration option dependencies. For example, in the code change ID:265470, the developer clearly stated: “[...] *This is because the value for swift_store_config_file is set in the [DEFAULT] section instead of the [glance_store] section like it should be*” (Launchpad 2016b). These examples show that the identified issues pertain to already broken dependencies. We also notice that the release-candidate phase has the fewest number of issues raised. This could be attributed to the time constraints faced by developers and reviewers, who may be under deadline pressure to successfully launch the new OpenStack release. As a result, their focus tends to lean towards addressing major bugs that could potentially disrupt the timely release goal.

5 Study Implications

For researchers Our analysis of RQ1 demonstrates that the IaC component within the OpenStack ecosystem undergoes modifications throughout all three phases of the release cycle:

⁴³ <https://bugs.launchpad.net>

development, release candidate, and post-release. Researchers can delve deeper into the dependencies between IaC and Non-IaC changes to predict a change. Furthermore, this exploration can help identify best practices for IaC development depending on the different release cycle phases. Precisely, researchers are encouraged to investigate whether the changes occurring right before the release date are driven by last-minute optimizations, debugging efforts, or other factors. Furthermore, they are encouraged to delve into the nature of changes observed post-release and assess whether these changes are reactive responses to new issues or if they stem from a continuous improvement process. Besides, researchers are encouraged to build on top of our preliminary checklist to include new emerging IaC-related principles that are not covered in our study or those stemming from new advancements in the domain of IaC frameworks. While our checklist focuses primarily on what reviewers discuss with code change owners, future works can extend our study by exploring guidelines or best practices for writing code reviews while there are no established guidelines for IaC, surveys with expert IaC practitioners can help identify and document best practice for IaC implementation and design. Researchers are also encouraged to explore different factors of IaC code changes such as their inherent complexity, the number of dependencies with other code changes, etc. Moreover, our findings motivate further research on the underlying reasons for the observed patterns between IaC and Non-IaC code changes. For example, it is interesting to further investigate the development cycle that has a higher negative coefficient for IaC code changes duration compared to Non-IaC code changes. Understanding the reasons behind these patterns can shed light on the unique characteristics of IaC, such as its code change/review complexity, requirements, etc. Furthermore, researchers can apply our approach to pull-request based code review processes to investigate whether the observed patterns between IaC and Non-IaC changes hold true in these platforms, and to potentially uncover platform-specific or project-specific influences on code review attributes that will enrich our understanding of IaC.

For practitioners Practitioners can use the results of RQ1 in the IaC modifications occurring throughout the release cycle to better understand and manage changes in their IaC components. This knowledge can guide development practices and help in anticipating/predicting potential challenges at different cycle phases. Besides, the parallel evolution observed between IaC and Non-IaC code changes underscores the importance of collaborative efforts among development teams and reviewers. A collaborative and effective code review process should be tailored to accommodate both IaC and Non-IaC code changes. Furthermore, leveraging established code review practices for Non-IaC code can be beneficial for IaC code changes, especially when involving the same developers. Furthermore, our findings of RQ2 show that despite IaC scripts representing a minority percentage of the code base, their code changes require equal or more effort than Non-IaC code changes. Given that IaC code changes take as long as Non-IaC changes but involve a higher churn of more files, IaC practitioners should focus on managing IaC more effectively. Particularly, findings emerging from Catolino et al. (2019) show that configuration-related bugs take longer to be integrated in the code base. Therefore, we advocate future research to focus on effective IaC management, especially considering the challenges posed by configuration-related bugs. IaC scripts are found to be one part of the configuration system in OpenStack that can differ in the best development practices from one type of configuration file to another (Bessghaier et al. 2023). Therefore, identifying and involving the right individuals in the review and development process can significantly contribute to the overall stability and efficiency of the software project. Particularly, organizations can identify team members who demonstrate proficiency in IaC tools and code review practices, as it allows knowledge sharing between the development

team. Moreover, integrating these specialized developers into the development of Non-IaC code changes can maximize their contributions and ultimately align with organizational objectives for efficient infrastructure management.

For tool builders In the context of IaC, proactive measures can significantly enhance the quality of code reviews. IaC tool builders play a pivotal role in this process by incorporating automated checks that facilitate IaC code comprehensiveness and also assessment. One effective approach involves implementing bots equipped with integrated checklists, such as the one we propose in our study. This checklist, refined through a survey with Top reviewers involved in IaC code changes in OpenStack, encapsulates key considerations for evaluating various facets of IaC code reviews contributing to the overall quality of IaC code. Additionally, introducing specific linters tailored to capture deviations from established best practices proves invaluable. These linters should seamlessly integrate into the development workflow, automatically executing whenever a new IaC code change is on the verge of submission or has been submitted. This ensures that essential checks are consistently applied, mitigating the likelihood of developers who may occasionally forget to engage a linter for their code.

6 Threats to Validity

Internal Validity About the correctness of our checklist, we use the Krippendorff agreement score to confirm that both authors have identified the same criteria in each IaC code change, to help increase our confidence in the understanding of the identified criteria. Every disagreement is discussed with the co-authors until a consensus is reached. The initial agreement between the authors was measured using Krippendorff's α (Krippendorff 2018), yielding a score of 0.83, indicating strong agreement. After discussing any disagreements, the authors achieved a second-round agreement score of 0.99. Additionally, another threat to validity can be related to the sample we selected for manual analysis to identify our code review checklist, as it is not possible to analyze over 300 thousand code reviews. This threat to validity affects our experimental study related to the construction of our code review checklist (RQ4), as there could be other items that are relevant for the code review checklist that are not covered by our selected sample. To mitigate this issue, we selected a representative sample with a confidence level of 95%, and a confidence interval of 5% to select our sample. However, there may exist other code review items that are not covered by our representative sample. We believe an important future work is to expand our sample to potentially identify more items in our code review checklist. Hence, we do not claim that our checklist is comprehensive. Furthermore, our defined 38 items in our checklist might not be relevant for the three phases of the cycle. We acknowledge that while our checklist provides broad preliminary guidelines, certain items may not apply to or be present in every phase. Furthermore, we acknowledge that the topic of a code change could impact the review effort. For example, refactoring edits often prompt different review discussions than non-refactoring ones. As a future direction, we delve into the topics of IaC code changes. An additional threat relates to whether the assigned reviewers to a code change are all involved in the review process. This threat could overestimate the number of reviewers in RQ2. It is crucial to acknowledge that not every reviewer assigned to a code change necessarily leaves inline comments or exchanged messages. Some reviewers might just vote for the code review; some others could join the review process at a later time to thoroughly assess and validate or disprove the code change. Therefore, we consider the number of the assigned reviewers to a code change in the Gerrit platform as the estimated needed developers to review a code change, similar to other studies (AlOmar et al.

2021, 2022; Coelho et al. 2021; Rigby and Bird 2013). Furthermore, regarding our survey, we used a 4-point Likert scale to ensure that developers decide on whether an item in our checklist is considered relevant or not. Including a neutral option could potentially dilute the responses, as it would allow respondents to avoid making a definitive judgment, which might not align with our goal of validating our checklist. Furthermore, we acknowledge that the expertise of reviewers is a potential threat to the validity of our findings. To address this, we computed the Reviewer Contribution Score (RCS), revealing that the top-5% of reviewers mostly contributing to the OpenStack review process are reviewing IaC code changes. This finding supports the claim that experienced reviewers are involved in reviewing IaC changes.

Construct Validity The first threat to validity concerns the selection of IaC-related code changes. Our results might not accurately reflect all code changes that involve IaC scripts, as the code changes we studied were selected randomly from a set of code changes with at least one IaC file. Furthermore, we recognize that we might not find all co-modified files in a code change are dependent, i.e., explicitly refer to each other, which might overestimate the number of co-modified files for IaC code changes. We acknowledge that an IaC file may depend on the project's other IaC or Non-IaC scripts. However, we do not take these dependencies into account, as we focus on the files that are co-modified in a code change and are undergoing a related change. For example, in some IaC-related code changes, we found cases of release-notes or “.md” files co-modified with IaC scripts. These files are indeed not dependent but undergo the same change. Furthermore, other studies found that IaC and Non-IaC scripts are co-evolving (Jiang and Adams 2015), and based on our manual analysis, 98% of code changes included files that underwent the same change. Thus, we assume that all files co-modified in a code change undergo a related change. While we found that IaC-related code changes have larger sizes than Non-IaC-related changes in terms of the number of files, this could be a statistical effect evoked by the sampling procedure as the IaC-related changes should contain at least one IaC file. This threat could impact our selection of IaC and Non-IaC code changes. To further investigate this aspect, we manually analyzed a representative sample (confidence level = 95%, and confidence interval = 5%) of 379 code changes having at least one IaC file that was randomly selected using the `Sampling R` package.⁴⁴ Out of the 379 code changes, 291 contain only IaC scripts, while 88 included at least one Non-IaC file. These 88 code changes were manually analyzed to determine if the co-changed files were affected by an IaC-related change. From these 88 code changes, 63 code changes only contained additional release-notes or “.md” files, which were deemed IaC-related as they only document the committed changes. The remaining 25 code changes were further analyzed, where 13 were found to only include additional test files for the IaC-related changes. Finally, 12 code changes were left to examine, and among them, four were found to undergo no related impact to IaC, while the other eight did. Our analysis revealed that only four code changes out of the sample were not entirely related to IaC, resulting in a 98% precision rate. Another potential threat relates to external factors that could bias the comparison between the code review attributes for IaC and Non-IaC code changes. To mitigate this, we assessed the impact of the file's size, age, and release cycle on the nine attributes. Furthermore, the selection of our IaC code changes could overestimate the number of IaC reviewers among the pool of most active reviewers in RQ3. However, upon checking the share of code changes performed by both IaC and Non-IaC reviewers, we found that the top-5% of reviewers contributed more to Non-IaC code changes than to IaC code changes with a significant p-value <0.001 and

⁴⁴ <https://cran.r-project.org/web/packages/sampling/index.html>

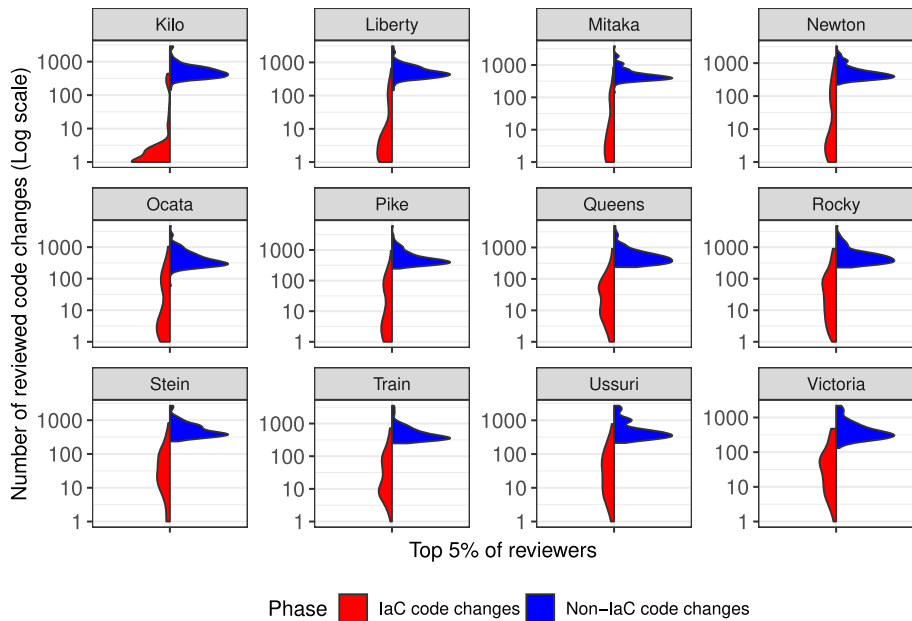


Fig. 13 The share of IaC and Non-IaC reviewed code changes by the top-5% reviewers

a large effect size as shown in Fig. 13. Consequently, the probability of a top-5% reviewer contributing to Non-IaC code changes is higher than for IaC code changes.

Another potential threat relates to the different review practices depending on the IaC tool. We did not differentiate code review practices based on the linguistic differences between Puppet and Ansible tools that could influence the review process. Our primary focus was to analyze the overall code review effort and patterns within the context of the IaC review in the OpenStack release cycle. However, examining the nuances between Puppet and Ansible could provide further insights into how specific syntactic and semantic characteristics might affect review practices and developers' contributions. Furthermore, a major threat to validity is related to the calculation of reviewers' experience. Obtaining the experience of each reviewer is challenging given the volume of data in our dataset, and that each code change could have more than one reviewer. Hence, we first removed all "services users" not to overestimate the reviewers' experience. Then, we adopted the Contribution Ratio (CR), used by prior research (Peruma et al. 2019; Scoccia et al. 2019; AlOmar et al. 2021) in the context of developer's contribution, where we used the number of reviewed code changes as a proxy of experience. The reasoning is that the more a reviewer participates in code reviews, the more experienced they become. Another potential threat relates to the use of the release dates as checkpoints at which the reviewer's contribution score is quantified. This threat can underestimate our evaluation of the reviewer's contribution scores in RQ3. We acknowledge that a reviewer contribution score should increase after every code change performed by the reviewer. However, since we have many code reviews (over 300k), it becomes impractical to compute the contribution score after every code change. Therefore, as we want to capture the engagement of reviewers with IaC code changes over time, we evaluate the contribution of reviewers on a release-by-release approach. We employ the release dates as checkpoints that will allow comparability between reviewers' contribution evaluation and showcase the

evolution of reviewers' contribution in IaC code changes over the releases. A possible threat is related to the pool of surveyed developers who validated our checklist. We acknowledge that these developers might not have a comprehensive background in IaC development principles. However, we addressed this threat by contacting the 100 most active developers (in terms of the number of performed code reviews) working on IaC code changes. Thus, we assume that these most contributing developers to IaC code changes are the most acknowledged of IaC development practices. Besides, we surveyed 9% of these developers, which falls within the accepted standards in software engineering (Khatoonabadi et al. 2023). Additionally, another threat relates to the possibility of surveying the OpenStack community to validate our checklist. We recognize that involving a broader segment of the OpenStack community could further enhance the validity and acceptance of the identified criteria. However, our decision to initially limit the validation to the top-100 developers was motivated by several factors. First, these top-100 developers possess a deep understanding of the IaC changes and the unique challenges associated with IaC, making their input particularly valuable. Second, the selection criteria for the surveyed individuals were based on their significant contributions and active participation in the review process, which we believed would ensure a high level of expertise and relevance in their feedback. Furthermore, we recognize that various factors may reflect a developer's expertise, including years of professional experience, frequency of contributions, and the diversity of projects. However, in this study, we specifically focus on a developer's review expertise (cf. Section 3.3.1), which we define based on the number of authored and reviewed code changes, in line with the approach by AlOmar et al. (2021). Our reviewer contribution score (RCS) represents how active a reviewer is in terms of reviewing and authoring code changes.

External Validity In this study, we focused on the OpenStack ecosystem because it extensively uses various IaC tools, which allows us to identify a wide range of IaC criteria. However, the diversity of IaC tools does not guarantee that our findings apply to other projects. The identified criteria may vary from one project to another. Despite this, the criteria we identified are generic and can be encountered in any IaC context. Still, researchers are encouraged to replicate our study using other software systems that adopt the IaC technology. However, the diversity of IaC tools considered in this study does not guarantee that our findings apply to other projects. Particularly, depending on the project's usage of IaC tools, the comparison between IaC and Non-IaC code changes may yield different results. Additionally, depending on the project's maturity, we might not find dedicated developers working on IaC code changes. However, the criteria identified in the checklist are generic and not project-dependent, meaning they can be encountered in non-OpenStack projects. A few of them are language-agnostic, such as *"Add filters to Ansible tasks for boolean comparison"*, which can be found in projects using Ansible IaC tool. Furthermore, our approach can be replicated in any code review platform, such as pull request-based projects, as these review platforms typically follow similar steps in the code review process. Furthermore, we acknowledge that our study focuses on merged IaC and Non-IaC code changes. We excluded abandoned code changes to reduce noise as they may have been abandoned for reasons unrelated to code quality (e.g., project changes, socio-technical or personal issues (Almarimi et al. 2023, 2020, 2021, 2020), developer priorities (Khatoonabadi et al. 2023; Wang et al. 2019), or accidental submissions (Wang et al. 2021a,b)). Therefore, we do not generalize our results to abandoned code changes, they represent another valuable area for further investigation and could deliver interesting insight. Hence, exploring the differences between merged and abandoned code changes, as well as the reasons for abandoning a code change, represents an interesting direction for future research.

7 Related Work

We present in this subsection related works on 1) IaC from different software engineering perspectives and 2) code review practices.

Infrastructure-as-code adoption, evolution, and maintenance One of the key benefits of using IaC is that it allows practitioners to apply the same code best-practices for managing infrastructure as they do for any other software application code. There have been several studies on this topic, but they have mainly focused on understanding good and bad practices in IaC coding (Kumara et al. 2021), IaC code smells detection (Rahman et al. 2019; Schwarz et al. 2018), and IaC quality assurance (Van der Bent et al. 2018; Dalla Palma et al. 2020; Sharma et al. 2016; Shambaugh et al. 2016).

Kumara et al. (2021) establishes a taxonomy of 10 types of best practices and four types of poor practices for three major IaC tools: Ansible, Chef, and Puppet. The analysis reveals the challenges and the solutions adopted by practitioners to address some of these challenges. The study concludes that more research is required on the development and maintenance of the IaC tools. Guerriero et al. (2019) further gained an understanding of the challenges related to the IaC development and testing in industrial environments through a survey of 44 practitioners. Jiang and Adams (2015) examined how IaC scripts and other software artifacts (such as source code and build files) co-evolve, and found that up to 45% of IaC changes tend to modify test or production files. Other researchers have focused on the maintainability of IaC code. For example, Van der Bent et al. (2018) proposed a maintainability measurement model for Puppet scripts. In the same vein, Dalla Palma et al. (2020) provided 46 metrics to evaluate Ansible IaC scripts, which covered (1) *long and complicated code* such as the number of blank lines and comments, (2) *resources overload* such as the number of commands and parameters, and (3) *Ansible practices* such as the number of external modules.

Similar to traditional software systems, researchers also investigate the detection of “*code smells*” in IaC scripts to evaluate their quality. Sharma et al. (2016) investigated poor Puppet IaC coding practices and proposed a catalog of 13 implementations and 11 designs smells. Rahman et al. (2019) examined the security aspect of Puppet files by identifying seven security code smells. Similarly, Schwarz et al. (2018) investigated code smells for the Chef IaC tool by identifying possible violations of Chef design guidelines. Shambaugh et al. (2016) proposed the *Rehearsal* tool to check the non-deterministic Puppet scripts. The determinism ensures the seamless execution of puppet files in different environments.

All the aforementioned studies have focused on investigating the best practices for coding with IaC. Our work builds on top of this research by focusing on code review best practices in the context of IaC configuration files.

Code review practices and impact The code review is among the effective code best practices that are extensively studied for code-related changes (Bacchelli and Bird 2013; McIntosh et al. 2016; Uchôa et al. 2020), but not for IaC configuration files. Several researchers investigated the relationship between code review and software quality. For instance, McIntosh et al. (2016) explored the impact of code review on the number of post-release defects from the perspectives of code review coverage, code reviewer degree of participation, and code reviewer expertise. Through a case study of four projects, the study fosters the intuition that poorly reviewed code could harm its quality. Uchôa et al. (2020) investigated the relationship between the review process’s problems and software design degradation. The results obtained on 14,971 code changes from seven software projects claim that the code review practices of long discussions and the high proportion of review disagreement increased the

design degradation. In the same vein, Pascarella et al. (2019) observed the impact of the code review practices on the severity of the code smells. Upon a sample of 21,879 code changes in seven open-source Java projects, the results suggested that code changes significantly influence the likelihood of reducing the severity of code smells. Furthermore, Pascarella et al. (2018) analyzed the questions and answers in the 900 code reviews of three Open source projects to identify the information reviewers need to conduct a proper code review. They found seven high-level needs, such as requesting a core reviewer to confirm the change or requesting further clarification of the context of the change. In the same line, Ebert et al. (2021) discussed the reasons behind reviewers' confusion in code reviews. Their analysis of manually analyzing code review comments and developers' responses based on a survey highlighted 30 reasons. Primarily, they found that long and complex code changes, unclear commit messages, and the dependency between different code changes are the Top three reasons leading to confusion.

Factors' impact on the code review process Recent studies have examined the impact of some factors on the code review process. Jiang et al. (2013) observed that the number of reviewers can impact the time taken for a code review. Building upon this, Bosu and Carver (2014) found that core developers receive faster initial feedback on their review requests and tend to complete the review process more expeditiously. Furthermore, Baysal et al. (2016) conducted an empirical study, revealing that both technical factors (e.g., patch size) and non-technical factors (e.g., author experience) may impact the duration of the code review process. Similarly, Thongtanunam et al. (2017) demonstrated that code review participation can be influenced by the author's past activities, the length of the change description, and the nature of the change being proposed. Notably, some studies reported that code review practices might vary depending on the files or practices that are being reviewed. For example, AlOmar et al. (2021) showed through an industrial case study at Xerox that reviewing refactoring changes takes a long review duration and requires more exchanged messages between the reviewers. Furthermore, Thongtanunam et al. (2015) showed that defective files are less rigorously reviewed than non-defective files. The study also shares the claim that the rigor of the review quality on a code file could impact its defect-proneness. In our study, we explore code review practice in the context of IaC scripts.

8 Conclusion

This study aims to understand the process of reviewing code changes for Infrastructure as Code (IaC). While code review is a common practice in both open-source and industrial projects, the practice of code review in the context of IaC has not been thoroughly studied. Therefore, we conducted a mixed-method study to explore the code review practices related to IaC and how they differ to Non-IaC code given their distinct operational and syntactical differences. First, we inspected and quantified the IaC code change review density based on ten review attributes. We found that reviewers take the same time to review and merge IaC code changes as they do for non-IaC code changes. However, IaC developers tend to generate 1.82 times more churn, while reviewers exchange 1.1 times more messages when reviewing IaC-related code changes to reach a consensus. Furthermore, we discovered that the top-5% of reviewers contribute more to the review process of IaC code changes than the remaining 95% of reviewers; that is, dedicated reviewers are specifically assigned to participate in the review of IaC code changes. Then, we qualitatively investigate the exchanged messages to identify the criteria reviewers rely on to accept merge an IaC code change. We found 38 criteria

within seven broad topics, such as *dependencies*, *compatibility*, and *logic*. Understanding the review practices used in IaC code changes can help managers more effectively plan for these reviews and allow developers to improve the quality of their IaC code changes by following best practices. This can also benefit reviewers by potentially reducing the duration required to review IaC-related code changes. It is also important for a developer to self-examine their IaC code changes to avoid coding issues that could slow down the review process. For future work, we plan to incorporate an analysis that differentiates code reviews based on the specific IaC tools used. This would involve examining whether the linguistic differences between IaC tools lead to variations in review time, the number of reviewers, types of issues encountered, and overall review practices. We plan to further support developers with which source files can be impacted in response to their related IaC changes. We also plan to explore IaC quality assurance tools and techniques that can support the code review process.

Author Contributions Narjes Bessghaier: Conceptualization, Methodology, Data curation, Validation, Formal analysis, Writing - original draft, Investigation. Ali Ouni: Conceptualization, Methodology, Validation, Writing - review & editing, Supervision. Mohamed Sayagh: Conceptualization, Methodology, Validation, Writing - review & editing, Supervision. Moataz Chouchen: Conceptualization, Methodology, Validation, Writing - review & editing. Mohamed Wiem Mkaouer: Conceptualization, Methodology, Validation, Writing - review & editing.

Funding This research has been partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) RGPIN-2018-05960.

Data Availability The datasets generated during and/or analyzed during the current study and the scripts are available in the following GitHub repository, <https://github.com/stilab-ets/iacreview>.

Declarations

Ethical Approval Not applicable.

Conflicts of Interest The authors declare that they have no conflict of interests and competing interests.

References

- Almarimi N, Ouni A, Chouchen M, Mkaouer MW (2021) csdetector: an open source tool for community smells detection. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1560–1564
- Almarimi N, Ouni A, Chouchen M, Mkaouer MW (2023) Improving the detection of community smells through socio-technical and sentiment analysis. *Journal of Software: Evolution and Process* 35(6):e2505
- Almarimi N, Ouni A, Chouchen M, Saidani I, Mkaouer MW (2020) On the detection of community smells using genetic programming-based ensemble classifier chain. In: Proceedings of the 15th International Conference on Global Software Engineering, pp. 43–54
- Almarimi N, Ouni A, Mkaouer MW (2020) Learning to detect community smells in open source software projects. *Knowl-Based Syst* 204:106201
- AlOmar EA, AlRubaye H, Mkaouer MW, Ouni A, Kessentini M (2021) Refactoring practices in the context of modern code review: An industrial case study at xerox pp. 348–357
- AlOmar EA, Chouchen M, Mkaouer MW, Ouni A (2022) Code review practices for refactoring changes: An empirical study on openstack pp. 689–701
- AlOmar EA, Liu J, Addo K, Mkaouer MW, Newman C, Ouni A, Yu Z (2022) On the documentation of refactoring types. *Autom Softw Eng* 29:1–40
- AlOmar EA, Peruma A, Mkaouer MW, Newman CD, Ouni A (2021) Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process* p. e2395
- AlOmar EA, Venkatakrishnan A, Mkaouer MW, Newman C, Ouni A (2024) How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations. In: 21st International Conference on Mining Software Repositories, pp. 202–206

- Alrubaye H, Mkaouer MW, Ouni A (2019) Migrationminer: An automated detection tool of third-party java library migration at the method level. In: IEEE international conference on software maintenance and evolution (ICSME), pp. 414–417
- Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 712–721. IEEE
- Baum T, Liskin O, Niklas K, Schneider K (2016) A faceted classification scheme for change-based industrial code review processes. In: 2016 IEEE International conference on software quality, reliability and security (QRS), pp. 74–85. IEEE
- Baysal O, Kononenko O, Holmes R, Godfrey MW (2016) Investigating technical and non-technical factors influencing modern code review. *Empir Softw Eng* 21:932–959
- Van der Bent E, Hage J, Visser J, Gousios G (2018) How good is your puppet? an empirically defined and validated quality model for puppet. In: 2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER), pp. 164–174. IEEE
- Bessghaier N, Sayagh M, Ouni A, Mkaouer MW (2023) What constitutes the deployment and run-time configuration system? an empirical study on openstack projects. *ACM Transactions on Software Engineering and Methodology*
- Bosu A, Carver JC (2014) Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In: Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement, pp. 1–10
- Brikman Y (2019) Terraform: Up & Running: Writing Infrastructure as Code. O'Reilly Media
- Catolino G, Palomba F, Zaidman A, Ferrucci F (2019) Not all bugs are the same: Understanding, characterizing, and classifying bug types. *J Syst Softw* 152:165–181
- Cliff N (1993) Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychol Bull* 114(3):494
- Coelho F, Tsantalis N, Massoni T, Alves EL (2021) An empirical study on refactoring-inducing pull requests. In: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–12
- Cohen J, Cohen P, West SG, Aiken LS (2013) Applied multiple regression/correlation analysis for the behavioral sciences. Routledge
- Conover WJ (1998) Practical nonparametric statistics, vol. 350. John Wiley & Sons
- Cruzes DS, Dyba T (2011) Recommended steps for thematic synthesis in software engineering. In: 2011 international symposium on empirical software engineering and measurement, pp. 275–284. IEEE
- Cuzick J (1985) A wilcoxon-type test for trend. *Stat Med* 4(1):87–90
- Dalla Palma S, Di Nucci D, Palomba F, Tamburri DA (2020) Towards a catalogue of software quality metrics for infrastructure code. *Journal of Systems and Software* p. 110726
- Dalla Palma S, Di Nucci D, Tamburri DA (2020) Ansiblemetrics: A python library for measuring infrastructure-as-code blueprints in ansible. *SoftwareX* 12:100633
- Davila N, Nunes I (2021) A systematic literature review and taxonomy of modern code review. *J Syst Softw* 177:110951
- Ebert F, Castor F, Novielli N, Serebrenik A (2021) An exploratory study on confusion in code reviews. *Empir Softw Eng* 26:1–48
- Edwards AL (1985) Multiple regression and the analysis of variance and covariance. WH Freeman/Times Books/Henry Holt & Co
- item example: C (2021a) <https://review.opendev.org/c/openstack/puppet-ironic/+528250>
- item example: C (2021b) <https://review.opendev.org/c/openstack/kolla-ansible/+676219>
- item example: C (2021c) https://review.opendev.org/c/openstack/kolla-ansible/+509186/1/ansible/group_vars/all.yml#319
- item example: C (2021d) https://review.opendev.org/c/openstack/tripleo-heat-templates/+238998/14..21/puppet/extraconfig/all_nodes/neutron-midonet-all-nodes.yaml#b89
- item example: C (2021e) https://review.opendev.org/c/openstack/kolla-ansible/+676216/2..3/COMMIT_MSG#b12
- item example: C (2021f) https://review.opendev.org/c/openstack/tripleo-heat-templates/+238998/2..21/puppet/manifests/overcloud_compute.pp#b71
- item example: C (2021g) https://review.opendev.org/c/openstack/tripleo-heat-templates/+269058/2..8/puppet/manifests/overcloud_controller_pacemaker.pp#104
- item example: C (2021h) https://review.opendev.org/c/openstack/puppet-octavia/+626637/9..12/spec/classes/octavia_worker_spec.rb#b54
- item example: C (2021i) https://review.opendev.org/c/openstack/tripleo-upgrade/+554914/4..5/templates/l3_agent_start_ping.sh.j2#b9
- item example: C (2021j) <https://review.opendev.org/c/openstack/puppet-tripleo/+613698/8..9/manifests/profile/base/tuned.pp#b22>

- item example: C (2021k) <https://review.opendev.org/c/openstack/openstack-helm-addons/+519591/6..8/.zuul.yaml#b17>
- item example: C (2021l) <https://review.opendev.org/c/openstack/bifrost/+649291/1/playbooks/roles/bifrost-ironic-install/tasks/main.yml#57>
- item example: C (2021m) <https://review.opendev.org/c/openstack/puppet-swift/+172021/5..7/templates/proxy/dlo.conf.erb#b1> (2021)
- item example: C (2021n) https://review.opendev.org/c/openstack/puppet-tripleo/+527403/6..7/manifests/profile/pacemaker/rabbitmq_bundle.pp#b229
- item example: C (2021o) <https://review.opendev.org/c/openstack/tempest/+509242/8..11/.zuul.yaml#b13> (2021)
- item example: C (2021p) <https://review.opendev.org/c/openstack/puppet-tripleo/+645477/3..4/manifests/selinux.pp#b48>
- item example: C (2021q) https://review.opendev.org/c/openstack/puppet-neutron/+307419/6..9/spec/classes/neutron_plugins_ml2_arista_spec.rb#b50
- item example: C (2021r) <https://review.opendev.org/c/openstack/puppet-neutron/+382551/2..6/manifests/plugins/ovs/opendaylight.pp#b76>
- item example: C (2021s) https://review.opendev.org/c/openstack/openstack-ansible-os_tempest/+625904/9..10/defaults/main.yml#b284
- item example: C (2021t) https://review.opendev.org/c/openstack/openstack-ansible-os_nova/+391532/5..6/tasks/nova_disable_smt.yml#b18
- item example: C (2021u) https://review.opendev.org/c/openstack/openstack-ansible-haproxy_server/+586774/4..12/tasks/haproxy_ssl_letsencrypt.yml#b60
- item example: C (2021v) https://review.opendev.org/c/openstack/openstack-ansible-haproxy_server/+586774/9..12/doc/source/configure-haproxy.rst#b156
- item example: C (2021w) <https://review.opendev.org/c/openstack/puppet-cinder/+150658/3..8/manifests/keystone/auth.pp#b103>
- item example: C (2021x) https://review.opendev.org/c/openstack/tripleo-heat-templates/+238998/20..21/puppet/manifests/overcloud_controller.pp#b317
- item example: C (2021y) https://review.opendev.org/c/openstack/openstack-ansible-os_barbican/+341761
- item example: C (2021z) <https://review.opendev.org/c/openstack/puppet-nova/+423507/1..4/manifests/metadata/novajoin/api.pp#b103>
- item example: C (2021aa) <https://review.opendev.org/c/openstack/puppet-neutron/+382551/5..6/manifests/plugins/ovs/opendaylight.pp#b76> (2021)
- item example: C (2021ab) https://review.opendev.org/c/openstack/puppet-cinder/+346531/1..4/manifests/volume/dellsc_iscsi.pp#b63
- item example: C (2021ac) https://review.opendev.org/c/openstack/tripleo-upgrade/+661302/25..33/tasks/upgrade/undercloud_os_upgrade.yml#b44
- item example: C (2021ad) <https://review.opendev.org/c/openstack/puppet-openstack-integration/+595370/3..4/manifests/repos.pp#b124>
- item example: C (2021ae) https://review.opendev.org/c/openstack/openstack-ansible-os_nova/+391532/1..6/tasks/nova_disable_smt.yml#b37
- item example: C (2021af) https://review.opendev.org/c/openstack/puppet-neutron/+307419/7/manifests/plugins/ml2/arista/l3_arista.pp#b66
- item example: C (2021ag) <https://review.opendev.org/c/openstack/tempest/+509242/6..11/roles/run-tempest/tasks/main.yml#b14>
- item example: C (2021ah) <https://review.opendev.org/c/openstack/puppet-tripleo/+674955/6/manifests/profile/base/docker.pp#306>
- Glaser BG, Strauss AL (1967) The discovery of grounded theory: Strategies for qualitative research. Routledge (1967). http://www.sxf.uuevora.pt/wp-content/uploads/2013/03/Glaser_1967.pdf
- Guerriero M, Garriga M, Tamburri DA, Palomba F (2019) Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 580–589. IEEE
- Han X, Tahir A, Liang P, Counsell S, Luo Y (2021) Understanding code smell detection via code review: A study of the openstack community. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), pp. 323–334. IEEE
- Hochstein L, Moser R (2017) Ansible: Up and Running: Automating configuration management and deployment the easy way. “O’Reilly Media, Inc.”

- Jiang Y, Adams B (2015) Co-evolution of infrastructure and source code-an empirical study. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pp. 45–55. IEEE
- Jiang Y, Adams B, German DM (2013) Will my patch make it? and how fast? case study on the linux kernel. In: 2013 10th Working conference on mining software repositories (MSR), pp. 101–110. IEEE
- Khatonabadi S, Costa DE, Abdalkareem R, Shihab E (2023) On wasted contributions: Understanding the dynamics of contributor-abandoned pull requests-a mixed-methods study of 10 large open-source projects. *ACM Transactions on Software Engineering and Methodology* 32(1):1–39
- Kim G, Debois P, Willis J, Humble J (2016) *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press
- Krippendorff K (2018) *Content analysis: An introduction to its methodology*. Sage publications
- Kula RG, Ouni A, German DM, Inoue K (2018) An empirical study on the impact of refactoring activities on evolving client-used apis. *Inf Softw Technol* 93:186–199
- Kumara I, Garriga M, Romeu AU, Di Nucci D, Tamburri DA, van den Heuvel WJ, Palomba F (2021) The do's and don'ts of infrastructure code: A systematic grey literature review. *Information and Software Technology* p. 106593
- Launchpad (2016a) Example of a code review bug on a package dependency issue. <https://bugs.launchpad.net/puppet-horizon/+bug/1556132>
- Launchpad (2016b) Example of a code review bug on configuration options conflict. <https://bugs.launchpad.net/tripleo/+bug/1532352>
- Li R, Soliman M, Liang P, Avgeriou P (2022) Symptoms of architecture erosion in code reviews: A study of two openstack projects. In: 2022 IEEE 19th International Conference on Software Architecture (ICSA), pp. 24–35. IEEE
- Likert R (1932) A technique for the measurement of attitudes. *Archives of psychology* (1932)
- McIntosh S, Kamei Y, Adams B, Hassan AE (2016) An empirical study of the impact of modern code review practices on software quality. *Empir Softw Eng* 21(5):2146–2189
- Moris K (2021) Infrastructure as code dynamic systems for the cloud age
- Napierala MA (2012) What is the bonferroni correction? *Aaos Now* pp. 40–41
- Opdebeeck R, Zerouali A, De Roover C (2022) Smelly variables in ansible infrastructure code: Detection, prevalence, and lifetime. In: *Proceedings of MSR'22: Proceedings of the 19th International Conference on Mining Software Repositories (MSR 2022)*. ACM
- OpenDev (2016) Example of a code review comment on a code review comment on best-practice issue. <https://review.opendev.org/c/openstack/puppet-neutron/+307419/6..9>
- OpenDev (2016) Example of a code review comment on ansible tasks dependency issue. https://review.opendev.org/c/openstack/openstack-ansible-os_nova/+391532/1..6/tasks/nova_disable_smt.yml#b37
- OpenDev (2016c) Example of a code review comment on security issue. <https://review.opendev.org/c/openstack/puppet-neutron/+404892/4..6/manifests/plugins/ml2/fujitsu/cfab.pp#b44>
- OpenDev (2017) Example of a code review comment on documentation issue: lack of explanatory code comments. https://review.opendev.org/c/openstack/kolla-ansible/+509186/1/ansible/group_vars/all.yml#319
- OpenDev (2017) Example of a code review comment on documentation issue: lack of release-notes. <https://releases.openstack.org/victoria/index.html>
- OpenDev (2017c) Example of a code review comment on iac guidelines ansible tasks issue. <https://review.opendev.org/c/openstack/tempest/+509242/8..11/zuul.yml#b13>
- OpenDev (2017d) Example of a code review comment on iac guidelines issue. <https://review.opendev.org/c/openstack/puppet-cinder/+491309/2..4/manifests/init.pp#b381>
- OpenDev (2018) Example of a code review comment on a configuration dependency issue. <https://review.opendev.org/c/openstack/puppet-openstack-integration/+595370/3..4/manifests/repos.pp#b124>
- OpenDev (2019a) Example of a code review comment on a compatibility issue. https://review.opendev.org/c/openstack/tripleo-upgrade/+661302/25..33/tasks/upgrade/undercloud_os_upgrade.yml#b44
- OpenDev (2019b) Example of a code review comment on logics issue. https://review.opendev.org/c/openstack/puppet-octavia/+626637/9..12/spec/classes/octavia_worker_spec.rb#b54
- OpenDev (2020) Example of a code review comment on testing issue. https://review.opendev.org/c/openstack/bifrost/+750656/comment/9f560f44_2663e843/
- OpenStack (2011) Diablo release. <https://releases.openstack.org/diablo/index.html>
- OpenStack (2016a) Branch model. https://wiki.openstack.org/wiki/Branch_Model
- OpenStack (2016b) Productteam. <https://wiki.openstack.org/wiki/ProductTeam>
- OpenStack (2019) Releases. <https://docs.openstack.org/contributors/common/releases.html>
- OpenStack (2020) Victoria release. <https://releases.openstack.org/victoria/index.html>
- OpenStack (2021) Release management. <https://docs.openstack.org/project-team-guide/release-management.html>


- OpenStack (2022) Stable branches. <https://docs.openstack.org/project-team-guide/stable-branches.html>
- Pascarella L, Spadini D, Palomba F, Bacchelli A (2019) On the effect of code review on code smells. *arXiv:1912.10098*
- Pascarella L, Spadini D, Palomba F, Bruntink M, Bacchelli A (2018) Information needs in contemporary code review. *Proceedings of the ACM on human-computer interaction* 2(CSCW):1–27
- Peruma A, Mkaouer MW, Decker MJ, Newman CD (2019) Contextualizing rename decisions using refactorings and commit messages. In: 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 74–85. IEEE
- Peruma A, Newman CD, Mkaouer MW, Ouni A, Palomba F (2020) An exploratory study on the refactoring of unit test files in android applications. In: *IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pp. 350–357
- Rahman A, Farhana E, Williams L (2020) The ‘as code’ activities: Development anti-patterns for infrastructure as code. *Empir Softw Eng* 25:3430–3467
- Rahman A, Mahdavi-Hezaveh R, Williams L (2019) A systematic mapping study of infrastructure as code research. *Inf Softw Technol* 108:65–77
- Rahman A, Parnin C (2023) Detecting and characterizing propagation of security weaknesses in puppet-based infrastructure management. *IEEE Transactions on Software Engineering*
- Rahman A, Parnin C, Williams L (2019) The seven sins: Security smells in infrastructure as code scripts. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 164–175. IEEE
- Rahman A, Rahman MR, Parnin C, Williams L (2021) Security smells in ansible and chef scripts: A replication study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30(1):1–31
- Ray B, Posnett D, Filkov V, Devanbu P (2014) A large scale study of programming languages and code quality in github. In: *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pp. 155–165
- Redhat (2022) What is infrastructure as code (iac)? <https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac>
- Rigby PC, Bird C (2013) Convergent contemporary software peer review practices. In: *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pp. 202–212
- Sadowski C, Söderberg E, Church L, Sipko M, Bacchelli A (2018) Modern code review: a case study at google. In: *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, pp. 181–190
- Saidani I, Ouni A, Mkaouer MW, Palomba F (2021) On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent. *Inf Softw Technol* 138:106618
- Sayagh M, Kerzazi N, Adams B (2017) On cross-stack configuration errors. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 255–265. IEEE
- Schwarz J, Steffens A, Lichter H (2018) Code smells in infrastructure as code. In: 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 220–228. IEEE
- Scoccia GL, Peruma A, Pujols V, Malavolta I, Krutz DE (2019) Permission issues in open-source android apps: An exploratory study. In: 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 238–249. IEEE
- Shambaugh R, Weiss A, Guha A (2016) Rehearsal: A configuration verification tool for puppet. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 416–430
- Sharma T, Fragkoulis M, Spinellis D (2016) Does your configuration code smell? In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pp. 189–200. IEEE
- Silva D, Tsantalis N, Valente MT (2016) Why we refactor? confessions of github contributors. In: *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, pp. 858–870
- Taylor M, Vargo S (2014) *Learning Chef: A Guide to Configuration Management and Automation*. “O’Reilly Media, Inc.”
- Teixeira JA, Karsten H (2019) Managing to release early, often and on time in the openstack software ecosystem. *Journal of Internet Services and Applications* 10:1–22
- Thongtanunam P, McIntosh S, Hassan AE, Iida H (2015) Investigating code review practices in defective files: An empirical study of the qt system. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pp. 168–179. IEEE
- Thongtanunam P, McIntosh S, Hassan AE, Iida H (2017) Review participation in modern code review: An empirical study of the android, qt, and openstack projects. *Empir Softw Eng* 22:768–817
- Turnbull J, McCune J (2011) *Pro Puppet*, vol. 1. Springer

- Uchôa A, Barbosa C, Oizumi W, Blenflío P, Lima R, Garcia A, Bezerra C (2020) How does modern code review impact software design degradation? an in-depth empirical study. 36th ICSME pp. 1–12
- Vasilescu B, Yu Y, Wang H, Devanbu P, Filkov V (2015) Quality and productivity outcomes relating to continuous integration in github. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering, pp. 805–816
- Wang Q, Xia X, Lo D, Li S (2019) Why is my code change abandoned? *Inf Softw Technol* 110:108–120
- Wang D, Kula RG, Ishio T, Matsumoto K (2021a) Automatic patch linkage detection in code review using textual content and file location features. *Inf Softw Technol* 139:106637
- Wang S, Bansal C, Nagappan N (2021b) Large-scale intent analysis for identifying large-review-effort code changes. *Inf Softw Technol* 130:106408
- Wang C, Lou Y, Liu J, Peng X (2023) Generating variable explanations via zero-shot prompt learning. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 748–760. IEEE
- Xu T, Zhou Y (2015) Systems approaches to tackling configuration errors: A survey. *ACM Computing Surveys (CSUR)* 47(4):1–41
- Yang X, Kula RG, Yoshida N, Iida H (2016) Mining the modern code review repositories: A dataset of people, process and product. In: Proceedings of the 13th International Conference on Mining Software Repositories, pp. 460–463

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations

Narjes Bessghaier¹ · Ali Ouni¹  · Mohammed Sayagh¹ · Moataz Chouchen¹ · Mohamed Wiem Mkaouer²

✉ Ali Ouni
ali.ouni@etsmtl.ca

Narjes Bessghaier
narjes.bessghaier.1@ens.etsmtl.ca

Mohammed Sayagh
mohammed.sayagh@etsmtl.ca

Moataz Chouchen
moataz.chouchen.1@ens.etsmtl.ca

Mohamed Wiem Mkaouer
mwvmse@rit.edu

¹ ETS Montreal, University of Quebec, Montreal, QC, Canada

² Rochester Institute of Technology, Rochester, NY, USA

Terms and Conditions

Springer Nature journal content, brought to you courtesy of Springer Nature Customer Service Center GmbH (“Springer Nature”).

Springer Nature supports a reasonable amount of sharing of research papers by authors, subscribers and authorised users (“Users”), for small-scale personal, non-commercial use provided that all copyright, trade and service marks and other proprietary notices are maintained. By accessing, sharing, receiving or otherwise using the Springer Nature journal content you agree to these terms of use (“Terms”). For these purposes, Springer Nature considers academic use (by researchers and students) to be non-commercial.

These Terms are supplementary and will apply in addition to any applicable website terms and conditions, a relevant site licence or a personal subscription. These Terms will prevail over any conflict or ambiguity with regards to the relevant terms, a site licence or a personal subscription (to the extent of the conflict or ambiguity only). For Creative Commons-licensed articles, the terms of the Creative Commons license used will apply.

We collect and use personal data to provide access to the Springer Nature journal content. We may also use these personal data internally within ResearchGate and Springer Nature and as agreed share it, in an anonymised way, for purposes of tracking, analysis and reporting. We will not otherwise disclose your personal data outside the ResearchGate or the Springer Nature group of companies unless we have your permission as detailed in the Privacy Policy.

While Users may use the Springer Nature journal content for small scale, personal non-commercial use, it is important to note that Users may not:

1. use such content for the purpose of providing other users with access on a regular or large scale basis or as a means to circumvent access control;
2. use such content where to do so would be considered a criminal or statutory offence in any jurisdiction, or gives rise to civil liability, or is otherwise unlawful;
3. falsely or misleadingly imply or suggest endorsement, approval, sponsorship, or association unless explicitly agreed to by Springer Nature in writing;
4. use bots or other automated methods to access the content or redirect messages
5. override any security feature or exclusionary protocol; or
6. share the content in order to create substitute for Springer Nature products or services or a systematic database of Springer Nature journal content.

In line with the restriction against commercial use, Springer Nature does not permit the creation of a product or service that creates revenue, royalties, rent or income from our content or its inclusion as part of a paid for service or for other commercial gain. Springer Nature journal content cannot be used for inter-library loans and librarians may not upload Springer Nature journal content on a large scale into their, or any other, institutional repository.

These terms of use are reviewed regularly and may be amended at any time. Springer Nature is not obligated to publish any information or content on this website and may remove it or features or functionality at our sole discretion, at any time with or without notice. Springer Nature may revoke this licence to you at any time and remove access to any copies of the Springer Nature journal content which have been saved.

To the fullest extent permitted by law, Springer Nature makes no warranties, representations or guarantees to Users, either express or implied with respect to the Springer nature journal content and all parties disclaim and waive any implied warranties or warranties imposed by law, including merchantability or fitness for any particular purpose.

Please note that these rights do not automatically extend to content, data or other material published by Springer Nature that may be licensed from third parties.

If you would like to use or distribute our Springer Nature journal content to a wider audience or on a regular basis or in any other manner not expressly permitted by these Terms, please contact Springer Nature at

onlineservice@springernature.com