Noname manuscript No.
(will be inserted by the editor)

An Empirical Study on Cross-component Dependent Changes

A Case Study on the Components of OpenStack

Ali Arabat · Mohammed Sayagh

Received: date / Accepted: date

Abstract Modern software systems are composed of several loosely coupled components. Typical examples of such systems are plugin-based systems, microservices, and modular software systems. Such types of software systems have several advantages motivating a large body of research to propose approaches for the migration from monolithic software systems to modular architecture (mainly microservices). However, a few prior works investigated how to assist practitioners post-migration. In fact, these studies reported that having independent components is difficult to achieve, leading to several evolution challenges that are still manually handled. In this paper, we conduct an empirical study on OpenStack and its 1,310 projects (aka., components) to better understand how the changes to a given component depend on changes of other components (aka., cross-component changes) so managers can better plan for their changes in a cross-component project, and researchers can design better solutions to help practitioners in such a co-evolution and the maintenance of multi-component software systems. We observe that the concept of ownership exists in the context of OpenStack, as different teams do not share the responsibility over the studied components of OpenStack. Despite that, dependencies across different components are not exceptional but exist in all releases. In fact, we observe that 52,069 OpenStack changes (almost 10% of all the changes) depend on changes in other components. Such a number of cross-component changes continuously increased over different years and releases, up to a certain release in which OpenStack decided to make a major refactor-

Ali Arabat
ÉTS - Québec University
Montréal, Canada
E-mail: ali.arabat.1@ens.etsmtl.ca

Mohammed Sayagh
ÉTS - Québec University
Montréal, Canada
E-mail: mohammed.sayagh@etsmtl.ca

ing of its project by archiving over 500 projects. We also found that a good percentage of cross-component changes (20.85%) end up being abandoned, leading to wasteful synchronization efforts between different teams. These dependent changes occur for different reasons that we qualitatively identified, among which configuration-related (34.64%) changes are the most common, while developers create cross-component changes for testing purposes then abandon such changes as the most prevalent category (38.45%). These cross-project changes lead to collaboration between different teams to synchronize their changes since 24.55% of the pairs of two cross-component changes are made by different developers, while the second change is reviewed by the developer of the first change of the pair (71.63%). Even when a developer makes both changes, that developer ends up working on a project that she/he is less familiar with. Our results shed light on how different components end up being dependent on each other in terms of their maintenance, which can help managers better plan their changes and guide researchers in proposing appropriate approaches for assisting in the maintenance of multi-component systems.

Keywords Empirical Study · Modularity · Software Evolution · Software Maintenance

1 Introduction

Modern software systems are composed of multiple components (aka, multi-component systems). Typical examples of these systems are plugin-based systems (e.g., WordPress [20]), Service-Oriented-Architecture [1], and Microservices [23]. Such software systems are organized as a collection of small and loosely-coupled components, each of which is developed, maintained, and deployed independently from each other [23]. In fact, in such systems, different teams are expected to work independently on their respective components without synchronizing with other teams working on other components [23]. Such loosely coupled types of systems lead to a key benefit which is related to the time-to-market (i.e., rapid and safe delivery of the software product [23]). In fact, when one component undergoes changes based on the requirements needs, only that component is expected to be deployed into production without impacting the processes of the remaining running services. For instance, all high-tech companies such as Uber, Amazon, and Netflix implement such a type of software architecture [11, 13, 33].

Another well-known example of multi-component software systems is OpenStack. It is a popular cloud computing platform that offers a wide range of components to set up a cloud-based infrastructure. Among these components is "Nova", which is a computing service used to create and manage virtual machines and bare-metal servers. "Swift" is another component for storing and retrieving massive data at scale. The diversity and flexibility of these components make OpenStack one of the most popular and powerful cloud computing solutions in the market. As a matter of fact, it is one of the three biggest open-source projects [40] where it is deployed on over 40M nodes around the world, supported by over 560 leading software companies such as Redhat, and accounting for more than 9,000 community contributors [50].

To benefit from the advantages of multi-component architectures, microservices in particular, several studies [14, 22, 25] tried to help practitioners migrate from monolithic to microservice systems. For example, Fan et al. [14] proposed a process to migrate monolith-based mobile applications into a microservice-based architecture. Furda et al. [22] have discussed different techniques to migrate legacy enterprise systems to microservice wrt., typically encountered challenges (i.e., Multi-tenancy). Similarly, De Lauretis [25] developed a strategy that supports the migration from monolith to microservice architecture.

With these progresses in the literature, having independent components is difficult to achieve, leading to a tedious co-evolution of components, which is still a manual and non-straightforward activity [19, 24, 32]. Sampaio et al. [19] found that IBM has several evolution challenges, among which are the compatibility and consistency between microservice versions. For instance, they found that developers manually identify how their modifications impact other components/services and engage with these other services' developers to identify potential issues. Through a systematic grey literature review and interviews with developers from 10 different companies, a prior work [24] found that having a low coupling between services (aka., components) is difficult to achieve. Similarly, we found in a prior work [32] that incompatibility issues that occur between the plugins of a plugin-based system are common and harmful. These incompatibility issues are among the most dominant reasons why WordPress users do not upgrade the version of their plugins [1]. While prior studies point out that maintaining multi-component systems is challenging, no prior work exists on understanding the maintenance of such systems.

As a first step toward assisting practitioners in the evolution of their multi-component systems, we first aim to conduct an empirical study to understand the maintenance dependency (co-evolution) between different components. In particular, we wish to study the prevalence, characteristics, and impact of the co-evolution of different components. The impact is in terms of engaged practitioners since such a co-evolution engages different teams and in terms of wasted resources manifested as abandoned changes of the co-evolution between components. Our empirical results will motivate and guide future studies to develop approaches for assisting practitioners in the evolution of multi-component systems. Similarly to prior studies on the co-evolution between different artifacts [19, 24, 32, 43], our results will also help practitioners better estimate the required efforts for maintaining their multi-component systems.

To better understand the co-dependency/evolution of different components of OpenStack and shed light on the *prevalence*, *characteristics*, and *impact* of the co-evolution among different components, we conduct a quantitative and qualitative study on the changes of OpenStack and its 1,310 projects. In this paper, we focus on OpenStack since it has many components (over 1,310) and OpenStack respects the ownership concept as no component is shared

[1] https://wordpress.org/news/2009/10/plugin-compatibility-beta/

among different teams of OpenStack and even the projects that are maintained by the same team are maintained by different developers, as we found from a preliminary study (PQ). Thus, OpenStack is an ideal case study with a large number of components, each of which is maintained by different teams. Furthermore, a large body of research studied OpenStack in different domains, such as bug assignment [8], micro-collaboration [39], and infrastructure-as-code [45]. In particular, we address the following research questions:

RQ1. How prevalent is the co-evolution of different components? We observe that OpenStack has 52,069 cross-component changes, which is 9.7% of all the changes and 38.45% of changes with a dependency (either within/across components). Moreover, a change to one component may ripple through a large number of components up to 224.

RQ2. How does the prevalence of cross-component changes evolve? All OpenStack releases involve cross-component changes and such changes was increasing up to the "rocky" release, right after three releases ("stein", "train", and "ussuri"), they archived a large number (501) of projects. After which the number of cross-component (i.e., 2,151 and 863 were the number of cross-component changes in the "victoria" and "zed" releases, respectively) changes exhibited a gradual decrease. Yet, cross-component changes are still common in the recent releases of OpenStack.

RQ3. How often do cross-component changes end up being abandoned? A good percentage of cross-component changes end up being abandoned. In fact, 20.85% of the studied cross-component changes end up being abandoned. In addition, our findings indicate that 25.77% of the investigated cross-service dependencies have also been abandoned.

RQ4. What are the characteristics of cross-component changes? Our qualitative study has shown that the examined cross-component dependencies are related to different categories (10), while developers abandon such pairs of dependent changes for a variety of reasons (12). For instance, the *Configuration* category is the most dominant cause accounting for 34.64% of the qualitatively studied cross-component changes. Moreover, *Test* is the most widespread category for which developers abandon cross-component changes covering over a third (38.45%) of the studied changes.

RQ5. Who is responsible for addressing cross-component changes? While 24.55% of the examined cross-component dependencies were carried out by different developers, a large percentage (71.63%) of such developers often develop one change and participate in the review of the other change of a pair of cross-component changes. Moreover, just 8.65% of cross-component dependent changes were not reviewed by any shared developer, suggesting collaboration between different teams.

Our study ends with a set of recommendations to help practitioners in the maintenance and evolution of their components and guide researchers in studying cross-component dependencies. We recommend **managers** not to neglect cross-component changes as they occur frequently and over all the releases rather than exceptionally. We also shed light on when to expect such co-changes through qualitative analysis. We also recommend **managers** to select appropriate resources to develop two cross-component dependent changes as having the same developer on both changes can be slower and push the developer of both dependent changes to contribute to a project she/he is less familiar with. We also recommend **practitioners** not to consider migrating from a monolithic to a modular architecture as a lifelong solution, but restructure their architecture when the dependencies among components increase. Finally, our results suggest several take-home messages for **researchers**, among which is to develop mechanisms to identify impacted components by a given change ahead of time, leverage the causes of cross-component dependent changes that we identified in RQ4 to develop better solutions, identify what other components to test given a change to one component (according to RQ4) and identify when it is necessary to refactor a multi-component system to reduce the maintenance dependency.

To facilitate replication of our study, we provide a replication package [2].

The remainder of this paper is structured as follows: Section 2 sheds light on the background and related works about the co-evolution of multi-component systems. Section 3 provides an overview of the data collection and methodology. Section 4 provides answers to our research questions. Section 6 discusses the threats to the validity of our observations. Section 7 concludes the paper and discusses potential future research directions.

2 Related Work

While our study focuses on the maintenance of multi-component systems so they can better benefit from the advantages of modular architecture, the closest work to our study is hence the migration from monolithic to multi-component systems, in particular, microservices (Section 2.1), the evolution of multi-component systems (Section 2.2), the maintenance of a software's dependencies which can be seen as external components to a software system (Section 2.3), and finally the coupling evaluation in multi-component systems is reported in Section 2.4.

2.1 Migration from monolith to microservice architecture

Many research efforts [14, 15, 17, 19, 23, 25, 26, 32] have been performed on migrating a monolithic application to a microservice architecture, so a software system can better benefit from the modularity of the microservice architecture. For instance, microservice-based applications comprise a collection of small, independently maintained and deployable services. Each of these services is owned by a small team of developers; hence, communication between different teams is eventually reduced to ensure the safe and rapid release of the software product [23]. Mazlami et al. [17] developed a formal model to help

[2] https://github.com/aliarabat/OpenStack/tree/stable

with the microservice extraction process from an existing monolithic application. De Lauretis [25] leveraged a strategy of five different steps, starting from function analysis, through microservice creation, all the way to defining a microservice architecture. Sampaio et al. [19] proposed a service evolution model to support practitioners in the evolution of microservice-based systems. Research efforts have been devoted to help migrating a legacy system to a microservice-based on particular techniques, such as SDLC [14], and Strangler Fig pattern [32]. Furthermore, Gouigoux and Tamzalit [15] highlighted three major issues derived from an industry case study when migrating from monolith to microservice, namely, granularity, deployment, and orchestration perspectives. Similarly, Fritzsch et al. [26] presented intentions, strategies, and challenges during the microservice transition process based on a qualitative study on 14 systems across different domains, and 16 interviews with practitioners from 10 software companies. Abdellatif et al. [41, 34] proposed a taxonomy to modernize legacy object-oriented systems to SOA through a set of SE (i.e., service identification) techniques. Later, the authors reported through a survey with 45 practitioners the absence of automation tools supporting the transition from legacy systems to SOA [21]. They also built a strategy called "ServiceMiner" [31] which is a bottom-up code-based approach aimed at practically helping the identification of potential services in a legacy system. Similarly, Trabelsi et al. [41] developed a microservice identification technique named "MicroMiner" to support such a transition process. Finally, we refer to an existing systematic literature review for more details about the migration of monolithic systems to microservices [42].

Although this line of research suggested techniques, strategies, and models to support the transition from monolith systems to a distributed microservice architecture, such studies do not examine the maintenance of the multi-component systems in general and microservices in particular post-migration. In fact, according to prior studies [19, 24], practitioners still face challenges after migrating their software systems related to the interdependency between different microservices. Thus, our study complements this line of research as we consider the maintenance of the system already migrated.

2.2 Co-evolution of components in a multi-component system

The closest line of research to our paper studied the evolution of multi-component systems [19, 24, 32, 43]. Sampaio et al. [19] observed that most of the developers in IBM proceed with manual investigation to manage the evolution of different microservices versions, such manual task is challenging and error-prone. Similarly, Bogner et al. [24] reported that having independence between different components of a microservice-based system is still an ever-present challenge in the industry. As such, interviewees stated that ensuring low coupling and high cohesion between services is not a straightforward task. Similarly, Lin et al. [32] found that incompatibility issues during the co-evolution of the WordPress platform and its plugins are common. Hence, 32%

of such incompatibilities occur between WordPress and its plugins. Assunccao et al. [43] identified three major types of changes during the co-evolution of microservices which are *technical*, *service*, and *miscellaneous*. Thus, the authors concluded that most of the studied microservice-based systems often co-evolve in terms of technical matters (i.e., updating libraries) rather than business concerns.

Our work complements this line of research that pointed out the problem of the co-evolution of different components. For instance, our work considers this line of research as a motivation to empirically understand the co-evolution of different components of a multi-component system. In our study, we wish to understand the co-evolution between two components quantitatively and qualitatively, so managers can better plan their changes and researchers can better understand such a co-evolution so they can develop appropriate solutions that minimize the interactions between different components and/or predict it in advance.

2.3 Dependencies in software ecosystems

A large body of research studied the evolution and impact of dependencies in popular software package management systems such as NPM [16, 35, 38] and Maven [4, 5, 36]. Kula et al. [16] evaluated the effect of the so-called *micro-package* on the overall npm ecosystem. The authors reported that micro-packages are more frequent in the ecosystem and have a long chain of dependencies, hence, suggesting developers be aware of such potential issues. Cogo et al. [35] investigated the phenomena of dependency downgrades in the same ecosystem. Dependency downgrade (49%) is a common practice among developers according to the derived rationales (i.e., reactive and preventive). Similarly, Chowdhury et al. [38] found that trivial packages (i.e., packages implementing basic features) are being centrally adopted by a large number of projects, hence, they observe that 16% of the packages in the ecosystem are trivial packages. Such trivial packages can impart around 29% of the packages within the ecosystem. Relating to *maven* package manager, Raemaekers et al. [4] studied the practical use of *semantic versioning*. Therefore, roughly a third of the investigated library releases involve breaking changes. The authors recommend developers consider using semantic versioning as it provides convenient support when performing library upgrades. Jezek et al. [5] observed that incompatibility issues are common between a program and its libraries. Interestingly, Soto-Valero et al. [36] found that bloated packages (i.e., packages that are unnecessary in runtime) in maven are common. In particular, the authors reported that 15.4% and 57% of, respectively, inherited dependencies and transitive dependencies are declared bloated.

Our work differs from this line as we investigate how different components of the same system are co-maintained and how a change in one component depends on another change in another component still of the same software system. In fact, our focus is on horizontal dependencies between the components of the same software system, rather than a system and external artifacts that it uses.

2.4 Coupling Evaluation in multi-component systems

Researchers have also examined many different approaches to measuring the coupling degree between components within a multi-component system [10, 27, 48, 49]. Such coupling techniques can help practitioners quantify the dependencies among different components, which may pinpoint architecture refactoring strategies. To this end, Candela et al. [10] carried out a quantitative and qualitative study to better understand the main factors determining software remodularization. Hence, through an extensive study on 100 open-source projects and a survey with 29 developers, the authors found that the refactoring efforts based solely on cohesion/coupling are not sufficient to reach a high-quality remodularised system. Furthermore, Sousa et al. [27] investigated the evolution of the coupling, which is the level of dependence among modules. The authors identified that legacy classes (aka., classes that are introduced in the first version of the system) highly impact the coupling growth. Zhong et al. [48] proposed a coupling metric that measures the degree of coupling among microservices. For instance, their proposed coupling is calculated based on the number of interfaces (i.e., a class with a set of APIs) in the first microservice that are invoked by the classes of the other microservice and vice versa. Moreover, Li et al. [49] proposed another coupling technique, called organizational coupling, which measures the collaboration between developers belonging to different microservice teams. Such a coupling is mainly derived through microservice ownership (i.e., identifying microservice teams) and cross-service collaboration (i.e., how frequently developers contribute to the other microservices).

Even though the aforementioned studies proposed a variety of approaches to evaluate the coupling between different components, such coupling metrics are code-centric metrics. We instead focus on the coupling between components in terms of maintenance activities. We define two components are dependent if they co-evolve. These studies do not also focus on understanding the maintenance of different components, and how they evolve in practice.

3 Methodology

Since we study the co-evolution between different components, we need to identify a case study with multiple components with enough data to allow us to systematically identify related changes. Therefore, the objective of this section is to discuss our case study OpenStack (Section 3.1), how we identify different components of OpenStack (Section 3.2), how we identify the component associated with each change and related changes (Section 3.3), how we identify user bots (Section 3.4), and the metrics that we used to answer our research questions (Section 3.5). Note that each research question has its own approach subsection that further details how we answered each research question.

3.1 OpenStack as a case Study

To answer our research questions, we study OpenStack since it has an explicit way of declaring dependencies between different changes, including those that are related to different projects. We also use OpenStack, similar to a large number of prior studies [8, 9, 18, 28, 30], as a large-scale open-source software system. Finally, we also consider OpenStack as it has a large number of projects (aka., components) that are distributed over different services. For example, *Nova* is a compute service that has 12 projects, including a project called *nova*[3], *nova-powervm*[4], and *python-novaclient*[5]. Another example is *Swift*, an object storage service containing several related projects, such as *swift-bench*[6] and *swift-specs*[7].

3.2 Identification of the components of OpenStack

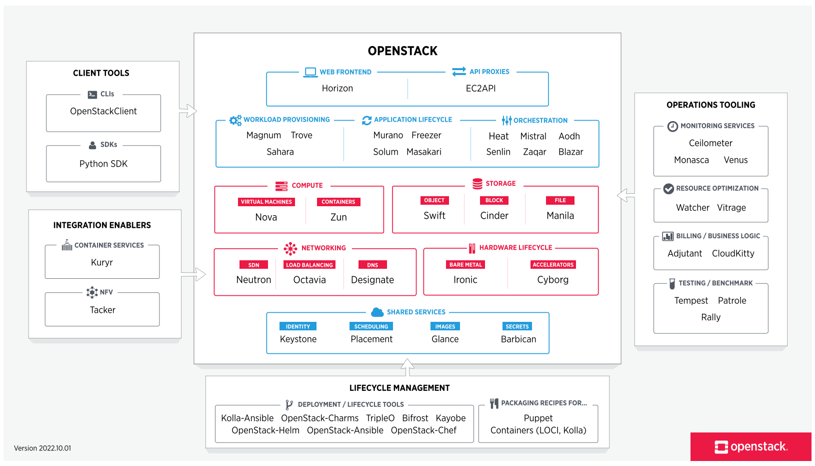


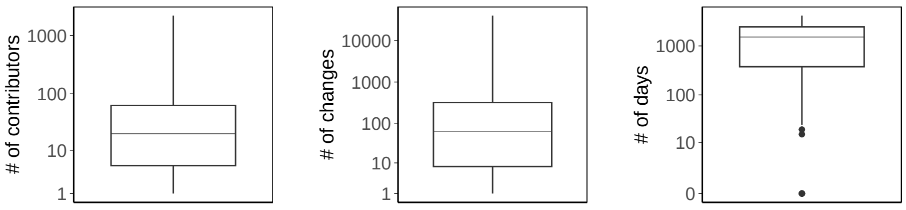
Fig. 1: A landscape of various OpenStack services obtained from OpenStack documentation homepage[8].

[3] https://opendev.org/openstack/nova
[4] https://opendev.org/openstack/nova-powervm
[5] https://opendev.org/openstack/python-novaclient
[6] https://opendev.org/openstack/swift-bench
[7] https://opendev.org/openstack/swift-specs

From the OpenStack documentation and the history of changes, we initially identified 61 services that together represent 1,310 distinct projects. Each of the 61 services has a team for which it is responsible, while that team has the same name as the service it is responsible for [9]. Note that 54 services are explicitly mentioned in the documentation as services [10] that are shown in Figure 1, while we derive the remaining seven services from the teams' names, such as the service *Requirements* which is a transversal service for configuring dependencies. Additionally, we refined the obtained services by removing *eight* services that do not have any change in Gerrit to avoid any misleading conclusions. We then extract the 1,310 projects from the history of changes in OpenStack [11]. Then, we map each project to its associated service by leveraging the following heuristic:

– **H1:** We first look at the project names that were modified by each team responsible for a service (i.e., note that a team has the same name for the service it is responsible for). For example, the projects *Openstack/nova* and *Openstack/os-vif* are two projects that belong to the *Nova* service [12] [13].

Our classification ends up with **53 services** and **587 out of 1,310 classified projects**. Note that the non-classified projects are excluded only from the cross-service-related changes and not from the cross-project changes. Cross-service changes represent two changes that belong to two completely different services among the 53 services. Cross-project changes represent the changes that belong to different projects among the 1,310 projects.



(a) Contributors. (b) # of changes. (c) Age distribution.

Fig. 2: Key characteristics of the studied OpenStack projects regarding the distribution of (a) **contributors**, (b) **changes**, and (c) **age** in days. All Figures were represented in logarithmic scale.

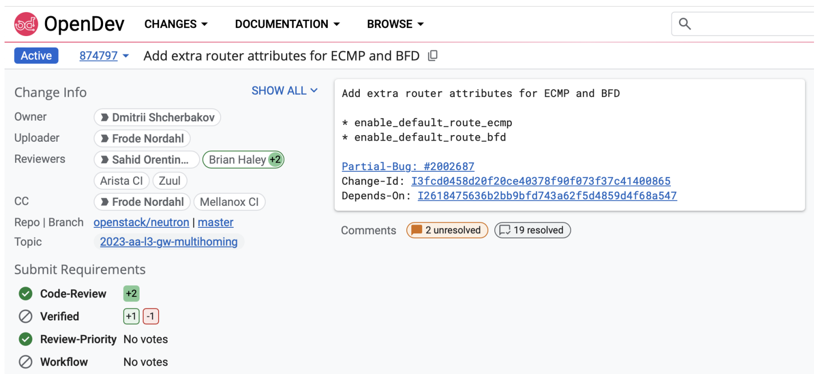
Our studied projects have a median of 20 human contributors (excluding bots), 63.5 changes, and a median age of 1,518.5 days, as shown in Figure 2. Our analysis covers a large number of changes amounting to 516,509.

8 https://www.openstack.org/software/
9 https://releases.openstack.org/index.html#teams
10 https://www.openstack.org/software/
11 https://releases.openstack.org/index.html
12 https://releases.openstack.org/teams/nova.html
13 https://governance.openstack.org/tc/reference/projects/nova.html

3.3 Related changes identification

To identify cross-project and cross-service related changes, we first map each change to its corresponding project and service, then identify changes that are related to each other.

3.3.1 Mapping changes to components



874797 Add extra router attributes for ECMP and BFD

Change Info

Owner	Dmitrii Shcherbakov
Uploader	Frode Nordahl
Reviewers	Sahid Orentin... Brian Haley +2 Arista CI Zuul
CC	Frode Nordahl Mellanox CI
Repo | Branch	openstack/neutron | master
Topic	2023-aa-l3-gw-multihoming

```
Add extra router attributes for ECMP and BFD

* enable_default_route_ecmp
* enable_default_route_bfd

Partial-Bug: #2002687
Change-Id: I3fcd0458d20f20ce40378f90f073f37c41400865
Depends-On: I2618475636b2bb9bfd743a62f5d4859d4f68a547
```

Submit Requirements

✓ Code-Review	+2
⊘ Verified	+1 -1
✓ Review-Priority	No votes
⊘ Workflow	No votes

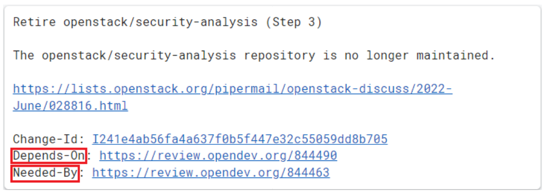
Fig. 3: An example of an OpenStack change with a set of information considered in our analysis[14].

We map each change to its corresponding project using the tag *"Repo"* as shown in the example in Figure 3. We then leverage the mapping of projects to their corresponding services using the mapping obtained following the approach discussed in Section 3.2.

To identify dependent changes, we leverage the following tags that are mentioned in the changes' description:

– **Depends-On:** is a tag that can be added to a change "A" to indicate on which other changes "A" depend. For example, the change in Figure 4a depends on the change *844490*. We apply the following regular expression `Depends-On:\s[a-zA-Z0-9/\.\:\+\-\#]{6,}` to extract the "Depends-On" id.
– **Needed-By:** is a tag that mentions that a change "A" is needed by another change "B". For example, the change in Figure 4a is needed by the change *844463*. Similarly, we leverage `Needed-By:\s[a-zA-Z0-9/\.\:\+\-\#]{6,}` regular expression to obtain the "Needed-By" id.
– **Related-Bug:** we also consider that changes for the same bug are related. To identify the bug to which the change is related, we leverage the "Related-bug" tag. To extract "Related-bug", we use the pattern `Related-Bug:\s#\d+`. Figure 4b shows a change related to bug *1174499*.

[14] https://review.opendev.org/c/openstack/neutron/+/874797



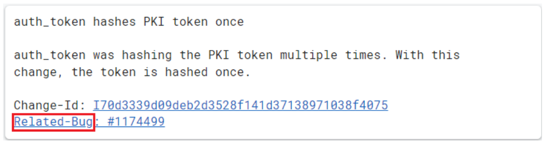
```
Retire openstack/security-analysis (Step 3)

The openstack/security-analysis repository is no longer maintained.

https://lists.openstack.org/pipermail/openstack-discuss/2022-
June/028816.html

Change-Id: I241e4ab56fa4a637f0b5f447e32c55059dd8b705
Depends-On: https://review.opendev.org/844490
Needed-By: https://review.opendev.org/844463
```

(a) An example of a change with the **Depends-On** and **Needed-By** tags



```
auth_token hashes PKI token once

auth_token was hashing the PKI token multiple times. With this
change, the token is hashed once.

Change-Id: I70d3339d09deb2d3528f141d37138971038f4075
Related-Bug: #1174499
```

(b) An example of a change with the **Related-Bug** tag

Fig. 4: Example of changes with tags that are used to identify dependent changes.

– **Change-Id:** we also consider two related changes if they share the same "Change-Id".

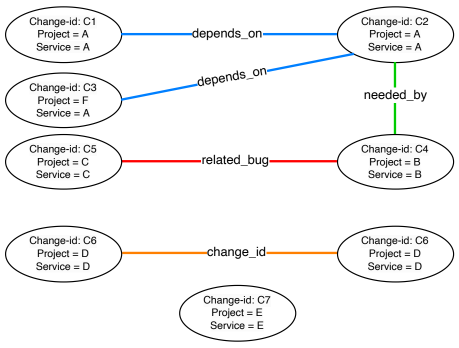


Fig. 5: An example of dependencies between OpenStack changes where nodes are changes made to components and edges are the relationships between them.

We then leverage the previous tags to build a graph of change dependencies and identify cross-project and cross-service related changes. An illustration of such a graph of dependencies is shown in Figure 5.

3.4 Bots Identification

Note that to identify whether a change owner within OpenStack is a human developer or a bot, we leverage an attribute called `tags` indicated in the user-related information. Therefore, a user bot is assigned the value `[‘‘SERVICE_USER’’]`.

Figure 1 provides a concrete example of the Zuul bot which automates a wide range of code review practices such as test execution and running gate jobs.

```
{
    "_account_id": 22348,
    "name": "Zuul",
    "username": "zuul",
    "tags": ["SERVICE_USER"]
}
```

Listing 1: A real example of a user bot related to Zuul.

3.5 Collected Data

To answer our research questions, we collect all the changes of OpenStack and extracted the data shown in Table 1. The same table also shows in which research question we used a given data, and we further detail how we answer each research question in the approach of the research question itself.

Table 1: Various information used to address each research question.

Information	Usage	Description
Number	RQ1-2-3-4-5	Unique ID of the change.
Project	PQ, RQ1-2-3-4-5	The project on which the change is made.
Team	PQ, RQ1-2-3-5	The team associated with the change.
Project type	PQ, RQ2	The type of the project (i.e., modified, non-modified, or archived).
Status	PQ, RQ1-2-3-4-5	The status of the change (i.e., open, abandoned, and merged).
Created	RQ2	The creation date of the change.
Insertions	RQ5	The number of inserted lines in the change.
Deletions	RQ5	The number of deleted lines in the change.
Reviewers	PQ, RQ5	The reviewers (account IDs) who participated in the review process of the change.
Messages	RQ4-5	The messages that were exchanged during the review process of the change.
Revisions	RQ5	Different revisions of the change.
Release	RQ2	The release on which the change was made.
Owner	PQ, RQ5	Original author (account ID) of the change.
Owner nature	PQ, RQ1-2-3-4-5	Whether the change is made by a human or service bot.
Commit message	RQ3-4	The commit message of the change.

4 Results

PQ. Do OpenStack teams have ownership over the components that they maintain?

Motivation: The goal of this preliminary research question is to empirically understand to which extent ownership of different services and projects exists in OpenStack. In particular, before investigating how different projects and services co-change and whether there is a need for approaches that help with the co-evolution of different projects/services, we first aim to investigate whether OpenStack teams have ownership over different OpenStack projects and services suggesting that these projects and services are expected to evolve independently. To do so, we will investigate whether there is an intersection between different OpenStack teams regarding the projects they are responsible for and modify, whether projects share developers, and whether developers and reviewers contribute to different projects or stick to one project.

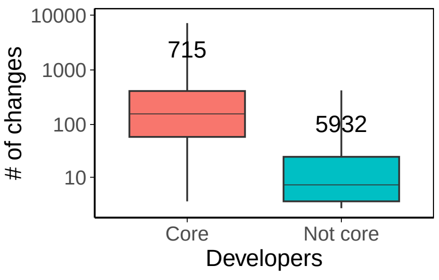
Approach: To investigate whether *different teams have ownership over different projects*, we extract the teams and the projects that each team modified in the past according to the documentation of OpenStack [15]. We then measure the intersection in terms of projects between each pair of teams. For example, we evaluate whether a pair of teams modified the same projects in the previous releases. The smaller the intersection between the teams, the more the ownership concept is respected in the context of the OpenStack projects, and these projects are expected to evolve independently.

As the previous analysis is to measure the intersection between different teams, we also measure whether *individual developers contribute to different projects* or whether they stick to the project their respective team is responsible for. In particular, we quantify for each pair of two projects the number of unique developers in each of the two projects and the number of developers who contributed to both projects. Similarly, we quantify for each pair of projects the number of reviewers who reviewed changes for just one of the two projects or both projects. We repeat the same experiment at the service level. For this last analysis, we collect all the changes of OpenStack as discussed in Section 3.3, the developers of these changes, the reviewers of these changes, and on which project such a change is made. We exclude casual contributors (defined below), so they do not bias our analysis. We also analyze whether core contributors are more likely to contribute to different projects than ordinary contributors, which are defined as follows:

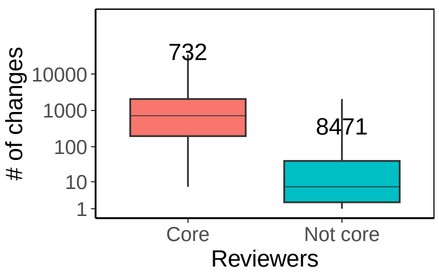
– **Casual contributors**: We exclude casual contributors from our analysis. Casual contributors are developers who have performed at most one change to OpenStack, as defined by Pinto et al. [12] and used by Batoun et al. [44]. We exclude such contributors from our analysis since they make just one change; hence, by definition, they never contributed to more than one OpenStack project. In other words, we exclude them to avoid biasing

[15] https://releases.openstack.org/index.html#teams

our findings on the percentage of developers who contributed to more than one project.
- **Core developers**: these are the main developers of a project as listed in the OpenStack documentation [16].
- **Contributors**: these developers have more than one change in OpenStack and are not part of the core developers.



(a) Number of OpenStack changes per core and non-core developers.



(b) Number of OpenStack changes each core and non-core developer has reviewed.

Fig. 6: The comparison of the distribution of OpenStack changes between core vs. non-core (a) **developers** and (b) **reviewers**. The number of individual developers is placed on top of boxplots.

Our dataset consists of 715 core and 5,932 non-core developers, and 732 and 8,471 core and non-core developers who reviewed changes. The core and non-core developers have a median of 157 and 7 OpenStack changes, respectively. Similarly, core and non-core reviewers participate in a median of 714 and 7 changes, respectively. Figure 6 shows the number of changes each developer and reviewer pushed/reviewed within OpenStack.

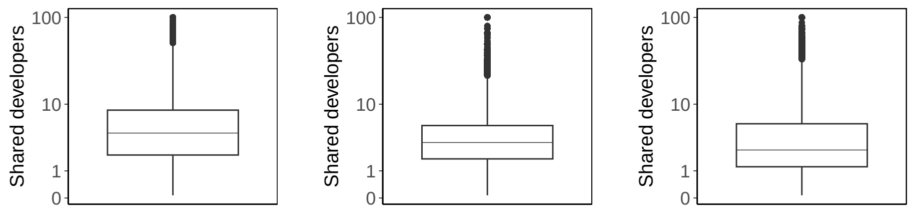
Table 2: The distribution of OpenStack projects among different releases notes' teams along with their intersection percentages.

Team pair	Common projects	Intersection (%)
Neutron - Octavia	*octavia* and *neutron-lbaas*	6.06%
Documentation - Oslo	openstack-doc-tools and openstackdocstheme	4.54%
Horizon - Manila	manila-ui	2.7%

Results: 99.76% (i.e., 1,972 out of 1,975 pairs) of the investigated pairs of OpenStack teams do not share any project. We observe that only three pairs of teams (shown in Table 2) share one to two projects. While these teams share projects, we observe from the documentation that these

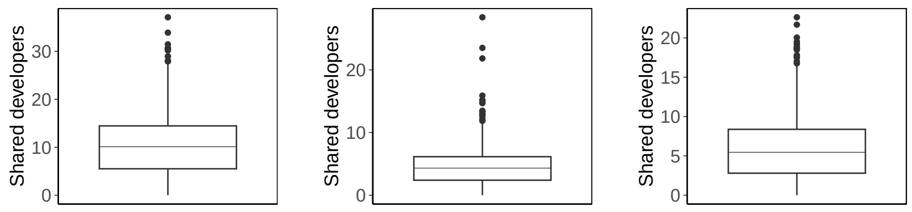
[16] https://review.opendev.org/admin/groups

pairs of teams did not release the same versions of their shared projects. For example, versions 0.5.0 to 0.5.2 and 0.9.0 to 0.9.2 of the “octavia” project were released by the “Neutron” team, while the “Octavia” team released other versions of the same project. Similarly, we observe that the “manila-ui” project was originally maintained by the “Horizon” team up to the OpenStack release “Mitaka”. Starting from the following release (i.e., “Newton”), the “manila-ui” project was maintained by the “Manila” team. The remaining shared projects show similar observations. Thus, even when two teams share a project, they do not maintain it simultaneously. One possible explanation is that OpenStack tried to promote the ownership principle of different teams over their corresponding projects. Consequently, only one team at a time is held responsible for the development of such a project [17] [18]. Our results suggest that the ownership of different teams over different projects exists for the OpenStack project.



(a) All developers. (b) Core developers. (c) Non-core developers.

Fig. 7: Distribution of developers’ intersection for (a) **all**, (b) **core**, and (c) **non-core** developers between the studied OpenStack pairs of projects.



(a) All developers. (b) Core developers. (c) Non-core developers.

Fig. 8: Distribution of developers’ intersection for (a) **all**, (b) **core**, and (c) **non-core** between the studied OpenStack pairs of services. Note that the intersections are defined in a percentage format.

[17] https://specs.openstack.org/openstack/designate-specs/specs/juno/zone-migration-between-tenants.html

[18] https://docs.openstack.org/keystone/latest/getting-started/architecture.html#projects

While 60.43% and 98.79% of our studied pairs of projects and services, respectively, have at least one developer who contributed to both projects or services of the pair, such an intersection across projects/services is not high. While 39.57% of our studied pairs of projects do not share any developers, we observe 1.19% of the studied pairs are developed by the same developers (i.e., both projects of the pair share 100% of the developers). By focusing on the 60.43% of the pairs of projects with at least one developer in common, we observe that these pairs share a median of just 2.83% of their respective developers (3.12% and 2.38% core and non-core, respectively), as shown in Figure 7. We observe similar results for the pairs of services. We observe that 98.79% (1,808 out of 1,830) pairs of services share at least one developer. For the pairs having an intersection in terms of developers, they share a median of 10.21% of their respective developers (a median of 4.38% and 5.59% of core and non-core developers, respectively), as shown in Figure 8.

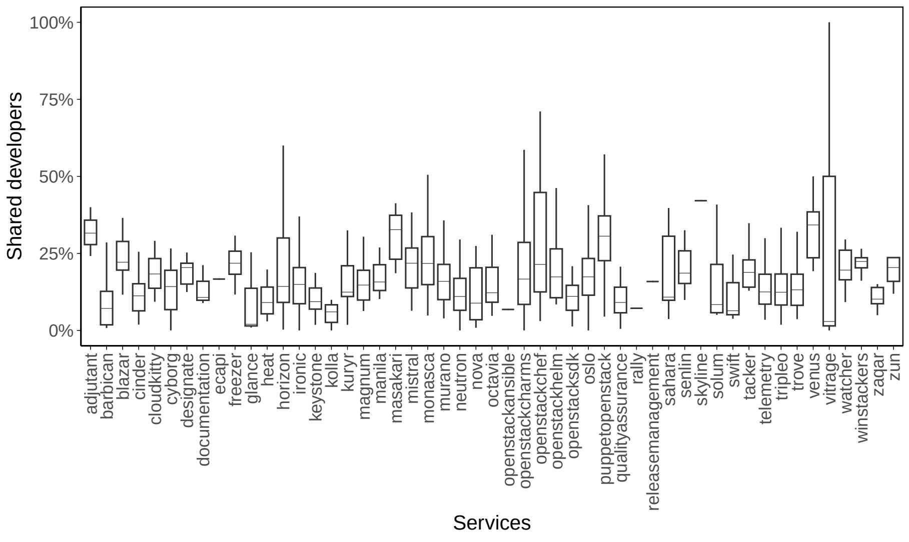
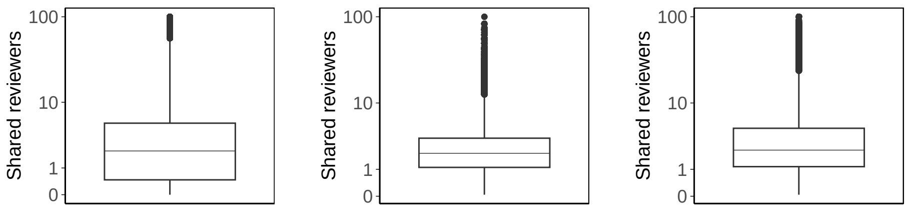


Fig. 9: Distribution of developers' intersection between pairs of projects within the same service. Note that four services ("requirements" and "shorlets") were not plotted because they contain only one project.

Additionally, projects of the same service have shared developers, but such an intersection is not high, as shown in Figure 9. The pairs of projects within the same service share a median percentage of developers as low as 2% ("glance" service) and as much as 42.10% (skyline service). In contrast, the maximum shared developers intersection is related to the "skyline" service, which can be expected since that service was just introduced and contains only two projects. In addition, we also observe that services related to logging ("venus"), admin ("adjutant"), and infrastructure ("puppet-openstack' and "masakari"') tasks involve a median of developers' intersection ranging from 30% to 34% as shown in the same figure. A potential explanation for such high numbers compared to other services is that these services are for configuration and transversal to the whole OpenStack, having potentially a high dependency on each other.

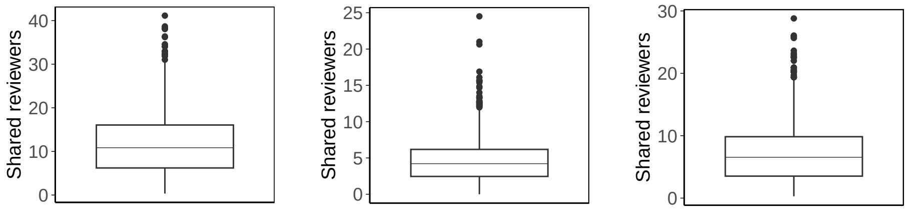


(a) All reviewers. (b) Core reviewers. (c) Non-core reviewers.

Fig. 10: Distribution of reviewers' intersection for (a) **all**, (b) **core**, and (c) **non-core** between the studied OpenStack pairs of projects.

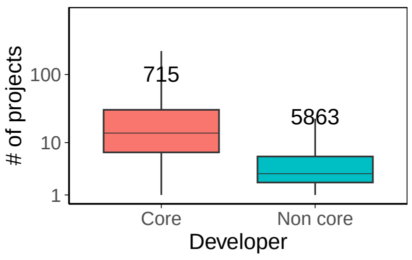
Our results hold for the reviewers as 78.77% and 100% of the pairs of projects and services have at least one reviewer in common, but such an intersection is not high. We observe that the projects with at least one reviewer in common have a median of 3.17% reviewers in common, among which are 2% and 2.22% core and non-core contributors as shown in Figure 10. Note that for core and non-core reviewers' intersection, we consider just the pairs of projects with at least one core and one non-core contributor, respectively, who reviewed changes for both projects. Similarly, the pairs of services with at least one reviewer share a median of 10.84% (4.35% and 6.54% core and non-core) of their respective reviewers as shown in Figure 11.

79.18% of the studied developers contribute to at least two OpenStack projects, with over half (62.56%) contributing to at least two services. On a median, developers participate in three different OpenStack projects. While 20.82% tend to focus on only one project, we observe that

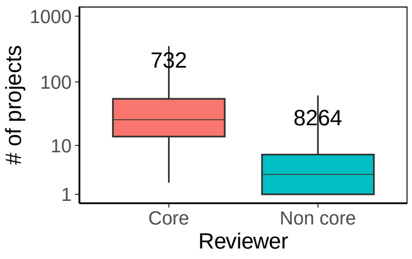


(a) All reviewers. (b) Core reviewers. (c) Non-core reviewers.

Fig. 11: Distribution of reviewers' intersection for (a) **all**, (b) **core**, and (c) **non-core** between the studied OpenStack pairs of services. Note that the intersections are defined in percentage format.



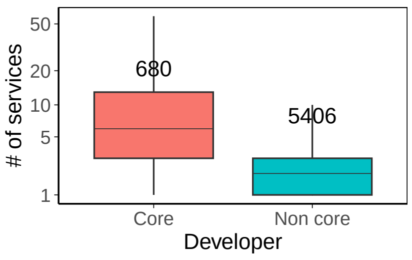
(a) Number of projects a developer contributes to.



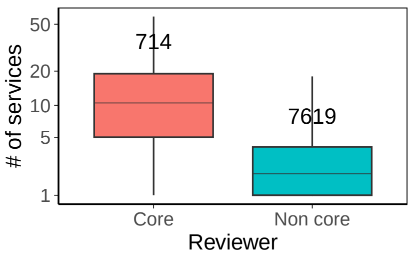
(b) Number of projects a reviewer contributes to.

Fig. 12: Number of OpenStack projects a (a) **developer** and **reviewer** contribute to, broken down into core and non-core sets. The number of individual developers is placed on top of boxplots.

a developer contributes to a maximum of 671 projects. The core developers have contributed to a median of 14 projects, whereas non-core developers have been involved in a median of three projects. Our 8,996 studied reviewers participated in a median of three projects. The core developers (732 reviewers) reviewed changes for a median of 26 projects, whereas the non-core reviewers (8,264 reviewers) were involved in a median of three OpenStack projects. These findings are depicted in Figure 12a and Figure 12b.



(a) Number of services a developer contributed to.



(b) Number of services a reviewer contributed to.

Fig. 13: Number of OpenStack services a developer and reviewer contribute to, broken down into core and non-core sets. The number of individual developers is placed on top of boxplots.

Our results hold for how developers and reviewers contribute to different services. Both developers and reviewers contribute to a median of two services. 40.8% (2,373 out of 5,815) of the OpenStack developers tend to focus on only one service. As it can be expected that core developers, with their large expertise, contribute to multiple services of OpenStack, we observed that they contribute to a median of five OpenStack services. Interestingly, even non-core developers contribute to a median of two services (Figure 13a), while less than half (44.23%) of non-core developers participate in only one service. We observe similar results for the number of services reviewers review. 40.62% of the reviewers focus on reviewing projects related to a single service. We also observe that core developers review a median of eight services and non-core developers review a median of two OpenStack services (as presented in Figure 13b). Figure 13 highlights the number of services each developer and reviewer participated in within OpenStack.

Summary of PQ

While the teams at a high level are responsible for different projects and projects do not share a large percentage of common developers and reviewers even for projects for the same service, developers and reviewers of these teams often contribute to multiple projects and services, creating a potential inter-dependency between these projects and changes in one project/service requiring changes in another one.

RQ1. How prevalent is the co-evolution of different components?

Motivation: While the preliminary research question shows that developers contribute to multiple projects, we aim by this research question to investigate if such dynamism is creating a changing dependency between different components. Quantifying these changes will help managers better understand the existence of changes, hence better planning for future changes and suggesting future studies to develop mechanisms to reduce such a dependency across different components to have better ownership of developers over their projects.

Approach: To quantify the dependencies across multiple projects/services, we first identify whether a change has a dependency on another change by leveraging the four tags (i.e., depends-on, needed-by, related-bug, and change-id) and methodology discussed in Section 3. Then, we map each change to its project (the Repo tag in Figure 3) to identify whether two dependent changes are related to the same project/service (aka., within-project/service changes) or they belong to two different projects (aka., cross-project/service changes). Thus, using these discussed tags and projects, we build a change dependency graph composed of changes represented as nodes and dependencies represented as the edges between nodes. A node contains the change-id and its corresponding repository. Figure 5 shows an example of our graph of dependencies.

From the graph, we identify the number of changes with a dependency to quantify the number of changes with a *dependency* on another project/service (aka., cross-project/service change). From our previous example of Figure 5, we have 7 out of 8 changes with a dependency. Among the seven changes, 4 changes (“C2”, “C3”, “C4”, and “C5”) are dependent on another component, and three (“C1” and and the two changes with “C6”) are single-component

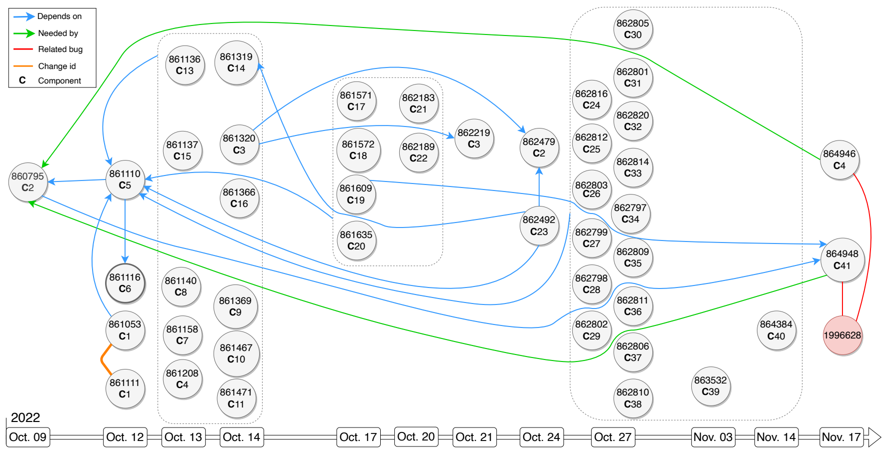
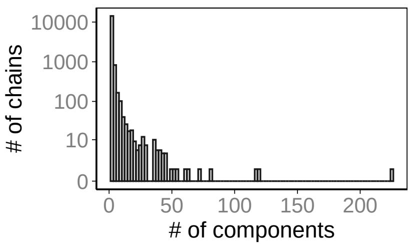


Fig. 14: A real example of a chain of 44 OpenStack dependent changes involving 40 distinct projects. Most of the relations among them are obtained through "Depends-On" tag (43 out of 48), then 2 for "Related-Bug" and "Needed-By", and "Change-Id" with only one relation. Note that nodes grouped inside a grey-dotted box share the same dependency. Red node is introduced bug.

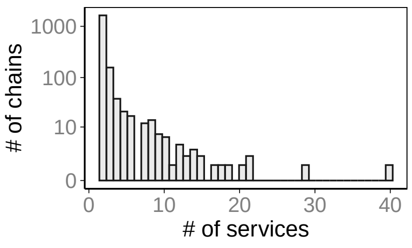
changes. From our mapping between project and services obtained following the approach discussed in our methodology (Section 3.2), we also count the number of cross-service changes. We have three cross-service and four within-service changes from our running example.

We also quantify the number of projects/services involved in a set of related changes (aka., chain of changes). Since the more projects involved in a set of related changes, the more synchronization and communications should take place; hence the more challenging that change is. Therefore, leveraging our graph of dependent changes, we also quantify the number of projects/services involved in a chain of related changes. For example, four projects (i.e., A, B, C, and F) are related to the chain of cross-component changes shown in our running example.

Results: 52,069 and 10,295 changes are cross-project and cross-service dependent, respectively. For instance, 23.49% (a total of 165,033 out of 537,387) of the whole changes of OpenStack depend on other changes. Among these changes, 31.5% (52,069 out of 165,033) of our studied changes depend on at least one change that belongs to a different project. 6.32% of the dependent changes cross the borders of different services, which might be more challenging as these dependent changes are across different services that provide different features, so they are less expected to be dependent, as is the case for projects of the same service. Figure 14 shows an example of a modification to the "devstack" project (which creates and initiates a full OpenStack environment) that resulted in an upgrade of the running Ubuntu release (from "focal" to "jammy") of its nodeset (i.e., a node typically runs an OpenStack service, e.g., compute node). This change required a change to the "tempest" project (an OpenStack testing framework) to update the version of its nodeset operating system to "jammy". Consequently, most of the changes (34 out of 44 changes highlighting a message like *"testing jobs on Ubuntu Jammy (22.04)"*), including tests, were performed against the change on "devstack" across various projects (e.g., "neutron").



(a) The number of OpenStack components per chain of dependent changes.



(b) The number of OpenStack services per chain of dependent changes.

Fig. 15: The number of chains of dependencies involving a certain number of (a) **projects** and (b) **services**.

The cross-project and cross-service dependent changes occur in a chain of dependent changes that can involve more than two projects in 20.7% of our studied chains and a median of 2 and 1 participating projects and services, respectively. 26.05% (15,467 out of 59,375) and 5.06% (3007 out of 59,375) of our studied chain of dependencies have at least one cross-project and one cross-service change dependency, respectively. While the majority (79.33%) of our studied chains of changes with a cross-project dependency involve two projects, as shown in Figure 15a, a chain can have up to 224 distinct projects. Similarly, we observe that 84.83% of our chain of changes with at least one cross-service dependency have two services and a maximum of 40 services, as shown in Figure 15b. Our results hint that dependent changes can involve multiple projects requiring further coordination between different teams.

Given the chain containing 224 OpenStack components (199 out of 231 dependent changes are merged), OpenStack has implemented a new project dependency management approach where the "requirements" team maintains and controls all versions of OpenStack libraries that each project may depend on. However, different OpenStack projects may use dependency versions different from each other, which could not be achieved by the global dependency configuration leveraged by the "requirements" project. Thus, based on these cross-component dependent changes, each project team had to create its dependency management file, called "lower-constraints.txt", containing the lowest versions of dependent libraries. Consequently, each project is, by now, more flexible in specifying the desired versions of libraries they depend on without impacting the others. Similarly, the same example involves the highest number of services, amounting to 42.

Summary of RQ1

We observe that cross-project and cross-service change dependencies are common (52k over 537k studied changes) leading to a chain of dependent changes with a median of two and up to 224 distinct projects. Hence, **OpenStack should consider dependent changes in their planning as a change to one component may propagate through several ones.**

RQ2. How does the prevalence of cross-component changes evolve?

Motivation: The goal of this research question is to investigate whether cross-project and cross-service dependent changes frequently occur over time and across releases, so managers should always book time and resources for such changes. Our findings will also shed light on whether such cross-project and cross-service change dependencies are getting more important over time, potentially leading to more interconnections between different projects and teams.

Approach: To determine how cross-project and cross-service dependent changes evolve over time and across releases, we leverage the cross-project/service changes (i.e., changes that are related to another change that belongs to another project/service) used to answer the previous research question. We identify the creation dates of the release related to each dependent change. To identify the release on which a given change is made, we look up if a change is made in a branch whose name starts with the "stable/" prefix followed by the release name or pick up the release issued right after that change if it is made on the main branch.

We quantify the evolution of changes over different years and different releases. We quantify the number of cross-project dependent changes created yearly from 2011 to 2022. We also quantify the number of dependent cross-project changes associated with each release. Similarly, we applied the same approach for cross-service changes over time and across releases. Note that for some dependent changes that are across different releases, we increment both releases since they both have a cross-project/service-dependent change.

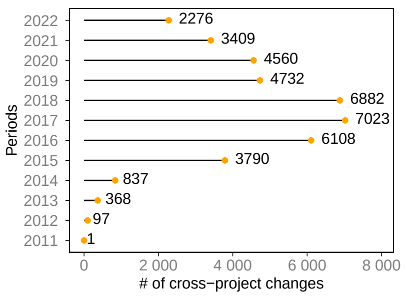
We also investigate whether there is a relation between the number of co-changes and the complexity of a release in terms of the number of modified projects in that release from one side, and the complexity of OpenStack in terms of the number of existing projects from the other side. In particular, we investigate whether more projects modified in a release are associated with a higher number of cross-project dependent changes. To do so, we compute the number of projects modified during each release according to the changes obtained from the OpenDev platform. We also examine if the number of archived projects in a release/year would lead to a decrease in cross-project changes. To draw such an analysis, we split projects into the three following categories:

– **Archived projects:** represent projects that are no longer maintained by OpenStack and have at least one change. Hence, they become dead projects. To identify archived projects, we download previously-archived OpenStack repositories from GitHub [19]. Each repository has an archival statement at the beginning of the page. Thus, we apply a regular expression pattern to retrieve the archival dates. We then transform the resulting dates into a proper date format for further processing. For instance, the "Bandit" archived project involves the following message at the beginning of the page in a yellow-background box: *"This repository has been archived by the owner on Jun 26, 2020. It is now read-only."*. We then filter projects that have at least one change.
– **Modified projects:** are the set of projects that were modified by OpenStack developers in a release/year. To identify modified projects, we leverage OpenStack changes in each year/release and the associated projects to each of these changes.
– **Non-modified projects:** indicates projects that are not modified in the current release/year, and modified at least once and not archived in the previous releases/years.

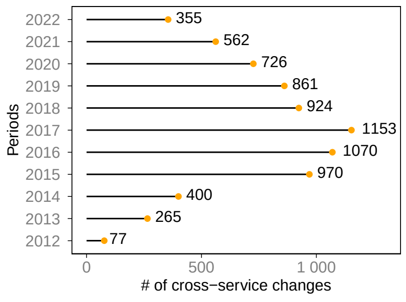
We also quantify the evolution of dependencies between projects, which is measured as the number of projects that a given project depends on across different years/releases. For example, project X depends on two other projects in the first release, whereas it depends on four projects in the second release, etc. To do so, we use dependent changes involving multiple projects and count the number of dependent projects associated with each project. For instance, a change X in component "A" and Y in another component "B", and both of them were created in the same year. Hence, "A" and "B" will each have one dependent project. Note that if X and Y were created in different years/releases, then they are not considered dependent projects. Additionally, we apply the same procedure for cross-project dependencies across releases, where only dependencies happening during the same release are counted. To achieve such consistency throughout this paper, only merged changes are included in our study.

Results: Cross-project and cross-service changes have grown superlinearly over the first five years of the OpenStack project. We observe that cross-project changes are gaining more importance over time. For example, the number of reported changes that depend on changes in other projects increased exponentially, particularly, between the 2011 (only one cross-project dependent change) and 2016 (6,108 cross-project changes) periods, as shown in Figure 16a. Afterward, we observed a gradual increase in the number of cross-project changes in 2017 (7,023) which recorded the highest rate, then remained fairly stable in 2018 (6,882). Over the following years,

[19] https://github.com/openstack-archive



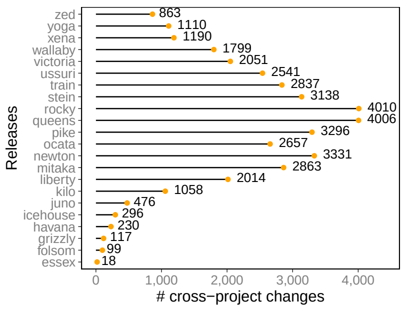
(a) Number of OpenStack cross-project changes over time.



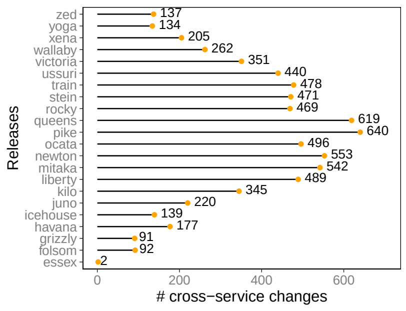
(b) Number of OpenStack cross-service changes over time.

Fig. 16: Evolution of (a) **cross-project** and (b) **cross-service** changes over time. Both figures follow the same trend.

we observe a slow drop in the number of cross-project dependent changes. We found similar patterns for cross-service changes over time, except in 2011 where one can observe no dependencies between changes involving different services, as illustrated in Figure 16b.



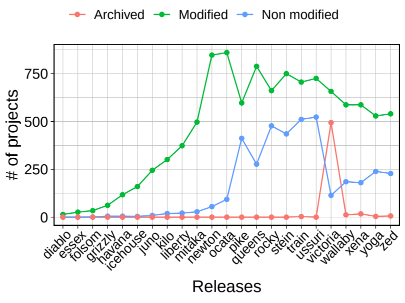
(a) Number of OpenStack cross-project changes across releases.



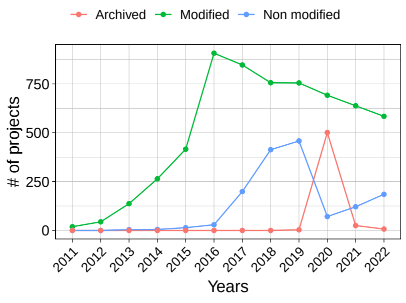
(b) Number of OpenStack cross-service changes across releases.

Fig. 17: Evolution of (a) **cross-project** and (b) **cross-service** changes across releases. The first four releases were omitted given the absence of related cross-project/service changes

Over its earliest stages, OpenStack has seen an explosive increase in the number of both cross-project and cross-service changes with "rocky" and "queens" releases having the highest proportion, respectively. We observe a steady growth of changes having dependencies with changes across multiple projects, as shown in Figure 17a. More precisely, between the "essex" release, which had 18 cross-project changes, and "icehouse" release with 296. Starting from the "juno" release, the number of cross-project dependent changes has been going through the roof up to the "newton" release (3,331), while it has remained less or more stable during the following two releases. The two subsequent releases namely, "queens" and "rocky", had 4,006 and 4010 (the highest proportion of recorded cross-project changes), respectively. Right after the "stein" release, we observe that cross-project dependent changes have gradually dropped off over the eight recent releases up to "zed" (which came out on October 5th, 2022). We observe similar trends for cross-service changes evolution across different OpenStack releases, except for some stable growths among certain pairs (e.g., "rocky" through "ussuri"), as shown in Figure 17b.



(a) The evolution of archived, modified, and non-modified projects across OpenStack releases.



(b) The evolution of archived, modified, and non-modified OpenStack projects over time.

Fig. 18: Evolution of different types of OpenStack projects (a) **across releases** and (b) **over time**.

The amount of cross-project changes is related to the load of changes in a given year/release as the more projects modified in each OpenStack release, the more cross-project changes are. Cross-project changes and the number of projects modified in each OpenStack release are highly correlated. As illustrated in Figure 18a, OpenStack has been continuously modifying more projects across releases, leading to a higher proportion of cross-project changes. Such correlation is confirmed through the Spearman rank correlation coefficient where the value marks a strong positive correlation ($\rho = 0.86$) for cross-project changes and continuously modified projects. For instance, the number of changed projects has doubled between "essex" (26 projects) and "grizzly" (62 projects) releases, producing 18 and 131 cross-project dependent changes, respectively. Such projects are constantly being changed until recording relative stability in "newton" and "ocata". Then, a slow decay is observed over the recent releases (e.g., "zed" had 540 modified projects). Additionally, one can still observe that archived projects were also a source of such cross-project changes decrease. For instance, Figure 18b shows the highest number of archived projects ever across all OpenStack history, recording 501 dead projects in 2020. In the following two years, OpenStack has archived 25 and 7 projects respectively. Consequently, Figure 16a reveals a noticeable decrease in the number of cross-project changes from 5,875 in

2020 to 4,335 in 2021. Moreover, cross-project changes and archived projects between 2019 and 2022 follow similar trends, as demonstrated in Figures 16a and 18b, respectively.

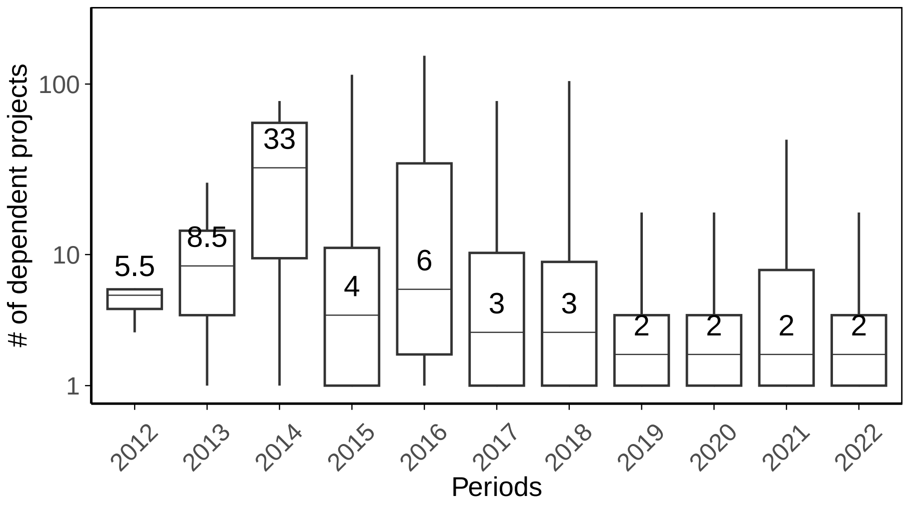


Fig. 19: Number of projects a given project depends on yearly. The medians are placed above the horizontal median lines.

The number of projects that an OpenStack project depends on has been steadily increasing but has remained relatively stable over the latest years and releases. Our observations suggest that OpenStack projects tend to rely more on the functionalities provided by other projects, resulting in higher levels of inter-project dependencies, as shown in Figure 19. For instance, in 2011, there was no change in one project depending on a change in another project (not shown in the figures). However, three years later, the number of dependent projects for a given project soared up to a median of 33 projects, representing the highest peak of the graph. Over the last couple of years (2015 to 2022), a project has had a median dependency on two to six other projects, indicating that OpenStack has reduced the potential connections between projects. Nonetheless, we observe that some projects still have a significant number of dependencies, with a maximum of 216 for “openstack-zuul-jobs” in 2018, while “project-config” was the most predominant in the following four stages from 2015 to 2017. Similar patterns were observed across different OpenStack releases, with noticeable stability in the number of dependent projects during the latest releases (a median of two dependent projects), especially from the “rocky” to “wallaby” releases, as depicted in Figure 20.

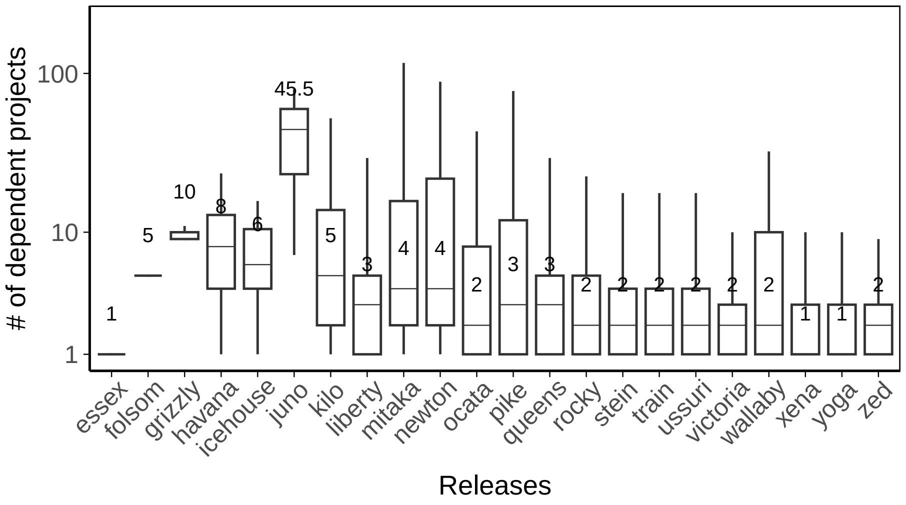


Fig. 20: Number of projects a given project depends on across releases. The medians are placed above the horizontal median lines.

Summary of RQ2

Over time and across releases, there has been a continuous increase in changes that depend on multiple projects and services within OpenStack. Additionally, the number of dependencies decreased after archiving a large number of projects. **Therefore, it is important for OpenStack to be mindful of these interdependencies when issuing new releases. Our results also shed light on the importance of restructuring a project even when it has a multi-component architecture**.

RQ3. How often do cross-component changes end up being abandoned?

Motivation: While RQ1 reveals that cross-project dependencies are common, the goal of this research question is to investigate to what extent such efforts end up being wasted as abandoned changes, similarly to prior work [46]. We assume that cross-project changes as they involve different teams might be more costly to be wasted as compared to with-component changes or changes that involve just one developer. Understanding the prevalence of such abandoned changes will motivate future studies to better deal with these types of changes.

Approach: Since abandoned changes can be a wasted effort according to prior work [46], we wish to quantify to which extent cross-component changes are wasted. To identify abandoned changes, we leverage a tag called status –shown in Figure 3 on the left hand of the change identifier. Such a tag can typically have one of the following statuses: *active*, *abandoned*, or *merged*. In this research question, we quantify the number of abandoned changes that are cross-component changes (i.e., have a dependency on a change in another component) according to changes that depend on other changes within the same component and changes without any dependency.

Results: The percentage of cross-project/service abandoned changes is three times the same percentage for within-project/service changes. Regarding the changes, the abandoned cross-project changes (20.85%) are three times greater than changes (7.9%) that depend on others within the same project, yet still lower than ordinary changes (21.9%) that do not depend on any other change. Similarly, we observe that abandoned cross-service changes (25.77%) are roughly three times that of within-service changes (9.71%). Related to the chains of dependencies, we observe that 4.12% (2,497 out of 59,375) of all studied chains of dependencies were found to be completely abandoned. Interestingly, the proportion of abandoned chains involving one project (20.79%) is twice that of chains of changes belonging to multiple projects (10.24%). On the other hand, we noticed that the percentage of abandoned chains related to cross-service dependencies (10.74%) is three times higher compared to within-service-related chains (3.06%).

Such an abandoned chain of dependencies contains at least two changes and a maximum of 35 related changes. For example, 34 changes depending on the change "210380" [20]. By abandoning "210380", all the other 34 changes were abandoned too, which would not be the case for ordinary changes with no dependencies. The change "210380" aimed to move a file (intall_modules.sh) from the root directory into the "tools" directory. However, the developers of this change found out it would trigger a large amount of work, so they eventually dropped it as well as all its 34 related changes, stating the following comment *"Yes, my preference would be to drop this patch unless there is a good reason for it."*.

Summary of RQ3

A good percentage of cross-project changes end up being abandoned, leading to a waste of effort on a whole chain of changes. Among the abandoned changes, some are designed to be abandoned for testing a change on other projects. **Hence, OpenStack should accordingly develop new mechanisms to detect abandonment of a cross-project change, so such time and effort are not wasted. Our results suggest future work to design approaches to prioritizing components to test for a given change**.

RQ4. What are the characteristics of cross-component changes?

[20] https://review.opendev.org/c/openstack/puppet-openstack-integration/+/210380

Motivation: This research question complements the previous ones through a deep qualitative understanding of the co-evolution among different components, such that practitioners can better plan for their changes in multi-component systems and research can design appropriate approaches to deal with the evolution of multi-component systems. To do so, we aim to conduct two qualitative analyses on (1) why cross-component changes occur and (2) why cross-component changes end up being abandoned. As we observe in RQ1 and RQ2 that cross-component changes are prevalent, we wish to understand the characteristics and the reasons behind these changes. As we also surprisingly observed in RQ3, practitioners abandon cross-component changes and as abandoned changes are wasted efforts according to prior work [46], we wish to understand to which extent it is the case for cross-component abandoned changes, whether practitioners need approaches to help them prevent abandoned changes, and why practitioners create a cross-component change that gets abandoned.

Approach: To qualitatively investigate why cross-component changes occur and why cross-component changes end up being abandoned, we select two randomly statistically representative samples (confidence level of 95% and confidence interval of 5%) of, respectively, **381 merged** and **364 abandoned** cross-component changes.

For both analyses, we went through each change (e.g., merged cross-component change for the 1st analysis and abandoned cross-project changes for the 2nd analysis) and investigated the change's title, description, and modified lines. If the relationship between a pair of changes is not clear enough, we, therefore, read through the discussion messages exchanged during the review process. The authors of this paper individually analyzed the 381 and 364 cases. After analyzing each case, the authors wrote a brief description (i.e., *"Add a text description on two files to warn the significance of libraries declaration order"*) about the reason for a cross-component change or abandoned cross-component change. Afterward, we performed **card sorting** technique (i.e., a commonly used technique when conducting qualitative analysis [3]) to classify each change under a given category. For the card sort technique for the 1st analysis (merged cross-component changes), we defined our labels during the card sorting technique, while in the 2nd analysis, we adopted the labels of Wang et al. [29] with a few differences. We used their categories since they studied abandoned changes in different projects, including OpenStack. As we observe during the labeling of some categories that can be further split into subcategories, we define the categories and sub-categories shown in Table 3 and Table 4.

Table 3: Classification of abandoned changes.

Category	Definition
Test	Changes that were designed for testing other changes and marked as not to be merged/reviewed.
Duplicate	Changes that were identical to other changes.
Lack of Feedback	Changes that did not have any activity for a long time.
Superfluous	Changes that were not needed at all.
Give Up	Developer gave up on finishing the change because it is time-consuming or laborious.
New Work	Developer of the change moved to another change.
Transfer	Changes that were transferred from one place to another.
Merge Conflict	Changes that were abandoned due to merge conflicts.
Contributor Operation	Changes that were abandoned because of the contributor's operation.
Complicated Change	Changes that were hard to resolve or need to be decomposed into smaller changes.
Incomplete/Wrong Fix	Changes that were incorrect/incomplete
Other	Changes that did not belong to any of the aforementioned categories.

Table 4: Subcategories of *Test*, *Duplicate*, *Transfer*, and *Contributor Operation*.

Category	Subcategory	Definition
Duplicate	Already Done	The problem was solved in another change.
	Integrated	Change was part of/integrated into another change.
	Suboptimal solution	Better solutions were proposed in other changes.
Transfer	Project	Change was transferred from one project to another.
	Branch	Change was transferred from one branch to another.
Operation	Wrong Project	Developer pushed the change into the wrong project.
	Wrong Author	The author of the change was wrongly assigned.

Table 5: The main reasons for relationships between two OpenStack cross-project dependent changes.

Reason	Description	Example	Count (%)
Configuration	Two dependent changes involve configuration tasks by adding, updating, and integrating config files, parameters, and gate jobs (i.e., Zuul pipeline) and using them in the second project, or even deleting the same config options from both projects.	The "499438" change modifies a config related file ("scenario001-multinode.yaml") which was used to execute a task in the "490376" dependent change.	**132 (34.64%)**

Continued on next page

Table 5 – *Continued from previous page*

Reason	Description	Example	Count (%)
Dependency	Dependencies also involve upgrading (i.e., upgrade a library–in the "requirements" project–to enable the use of newer functionalities in the 2nd project), removing (i.e., fixing a bug by removing an extension, or remove a no longer needed dependency), initializing an extension or library (i.e., add configuration options related to a project in the two projects, such as project name, version, and description...etc)	The "544025" change required a specific functionality only exists in the newer version of "gabbi" (i.e., 1.42.1) which forced the "requirements" project ("563173" change) to upgrade it globally.	73 (19.16%)
Code enhancement	Code improvement involve modification to code chunks and properties that are shared between two changes. Such quality tasks include fixing bugs, removing unsupported features...etc.	"473158" removes unnecessary code (i.e., control statement) where its business logic was implemented in the "473081" change.	48 (12.59%)
New features	New features often involve the implementation of API endpoints in the 1st project, hence consuming them in the 2nd project.	The "288392" exposes a set of API(s) while the "254280" dependent change implements its tests.	18 (4.78)%
Docs	This category is designed to provide documentation and more pieces of information to the end user by adding the same content in both dependent projects.	"118694" and "118729", which are cross-project dependent, stated the following message: "warn against sorting requirements", to warn users on the importance of the libraries order in which they are declared.	12 (3.14%)
Renaming	This category consists of renaming either the same files/source code in the dependent changes or property in the 1st change and exploiting it in the 2nd change.	The "620303" change renamed a config option (i.e., "tripleo-ci-centos-7-scenario001-standalone"), while the "619508" dependent change employed it in different places of changed files.	10 (2.62%)
Tests	This type of dependency implements tests-related modules and methods in the 1st change while consuming them in 2nd change.	The "281369" added a new module ("PuppetOpenstack-SpecHelpers") dedicated for testing purposes, whereas the "281376" change imported it within the changed files.	6 (1.57%)
Refactoring	Such a reason include refactoring the code across the dependent changes.	The "644929" and "659850" changes perform a wide range of refactoring such as reaming files and as the title of the later change indicates.	6 (1.57%)
Moving resources	This category involves moving files or code chunks between dependent projects.	A file called "test-install-haproxy.yml" was deleted from the "475040" change and added to the "476865" change.	5 (1.31%)
Others	Represent the instances that were not easy to identify the reason for their relationships.	N/A	71 (18.63%)

Results: The most frequent reason for cross-component changes is *configuration*, followed by *dependency management*, and *code enhancement*. Table 5 shows the primary reasons derived from the 381 qualitatively analyzed OpenStack cross-project dependencies. We observe that relationships between two dependent changes designed for configuration purposes are the most prevalent, accounting for 34.64%. For example, changes 536579 and 536588 involve the removal of the same configuration options (i.e., "tripleo-ovb-check" and "tripleo-ovb-experimental") from a configuration file ("layout.yaml") in both projects. *Dependency management* is the second most common reason for such a relationship, constituting 19.16% of the observed cross-project dependent changes. This type of relationship often involves tasks such as upgrading, removing, or initializing internal or third-party libraries. For instance, the introduction of a new project called "os-service-types" into the OpenStack ecosystem led the *487112* and *487177* dependent changes to, respectively, adding a file containing the project description, and adding the version of the same project to the global dependency management file ("global-requirements.txt"). Additionally, *code enhancement* is another widespread type of cross-project change representing 12.59%. The main purpose of this type involves improving existing code scattered across dependent changes. We observe a wide variety of rationales that are less frequent than the aforementioned categories, including *new features*, *docs*, *renaming*, *tests*, *refactoring*, and *moving resources* where each corresponds respectively to 4.78%, 4.14%, 2.62%, 1.57%, 1.57%, and 1.31% as shown in Table 5. However, 18.63% of the manually studied cross-project dependencies were classified into the others' category, making it difficult to determine the exact reason for the relationship between them.

Table 6: Various categories of abandoning OpenStack cross-project changes.

Category	Example	Count (%)
Test	Change *843091* was designed to test the Zuul job, hence mentioning the following description ("DNM: Testing tempest pin on stable/ussuri").	**140 (38.45%)**
Duplicate		50 (15.42%)
Already Done	Change *238222* upgraded the version of the "keystonemiddleware" library. After a while, the developer discovered that it had already been done in another repository leading to its abandonment.	29 (8.00%)
Integrated	Change *597291* was abandoned because it has been integrated into another change. Thus, the developer clearly states "*Moved to [600858]*".	14 (3.84%)
Suboptimal Solution	Change *864803* shows an instance of a developer abandoning it because a better solution was provided. The developer concluded such abandonment with the following message "*[864838] is the right one it seems*".	13 (3.58%)
Lack of Feedback	Change *250648* was abandoned by a core reviewer due to inactivity for more than three years. The core reviewer finished the change with "*Abandoning due to inactivity [...]*".	46 (12.63%)
Superfluous	Change *630736* took more than six weeks of review time. Right after, the developer found that such a change was irrelevant, which eventually ended up being abandoned and left the following comment "*Unnecessary changes committed*".	40 (11.00%)

Continued on next page

Table 6 – *Continued from previous page*

Category	Example	Count (%)
Give Up	Change *445681* was abandoned because its corresponding developer had no sufficient resources to keep working on it. As such, the change was concluded with the following message "*I haven't found resources to continue the efforts on it but I'll probably restore this work later*".	11 (3.02%)
New Work	Change *471479* was abandoned because the developer and reviewers agreed to move forward with another approach. Consequently, the developer concluded this "*After conversation on IRC, i will abandon the usage of delorean-repo and start using DIB_YUM_REPO_CONF*".	11 (3.02%)
Transfer		10 (2.75%)
Project	Change *310042* was transferred from *openstack/ironic* project to *x/ironic-staging-drivers* as "*moved to ironic_staging_drivers [325974]*" depicts.	8 (2.20%)
Branch	Change *810078* was transferred from *master* branch to *feature/r1* as also stated in the last comment "*Cherry-picked to feature/r1 as : [811951]*".	2 (0.55%)
Merge Conflict	Change *772099* was abandoned because there were merge conflicts between certain files as commented by the developer "*The following files contain Git conflicts: [ovsdb_monitor.py] and [requirements.txt]*".	5 (1.37%)
Contributor Operation		4 (1.09%)
Wrong Project	Change *299851* was submitted by its developer to the wrong project which resulted in its abandonment as follows "*Wrong project*".	3 (0.82%)
Wrong Author	Change *563970* was assigned the wrong author as this comment "*wrong author*" exhibits.	1 (0.27%)
Complicated Change	Change *100791* expresses the need to separate the concerns into multiple changes as one of the reviewers proposed the following "*Agree with [...] that the namespace wrapper should be in a separate change.*". After that, the same reviewer said that the current state of the change would be poorly developed as follows "*Try not to let random changes creep into your patch.*"	3 (0.82%)
Incomplete/Wrong Fix	Change *676814* was imperfect because the developer has set out a new direction to take as revealed in "*taking this change set another direction.*".	3 (0.82%)
Other	N/A	35 (9.61%)

OpenStack developers abandon cross-project changes for many different reasons, among which *Test* is the most widespread purpose for such abandonment. From Table 8, we observe that changes that were abandoned for *Testing* purposes are the most dominant (38.45%), followed by *Duplicate* changes (15.42%), then changes with a *Lack of Feedback* (12.63%). Additionally, changes that were abandoned, but not needed at all, are classified under the *Superfluous* category and are also common (11%). Interestingly, we notice a similar percentage of abandoned changes for which their respective developers gave up on finishing and continued the work on other changes which are, respectively, referred to as the *Give Up* and *New Work* categories. The other remaining abandonment categories, namely *Transfer*, *Merge Conflict*, *Contributor Operation*, *Complicated Change*, and *Incomplete/Wrong Fix* have percentage distribution values in the range of 0.82% to 2.75%. The

Other category involves abandoned changes that did not belong to any of the previously described categories. While Wang et al. [29] found that the most common reason for abandoning changes is duplicate changes, we find that it is the second category for cross-component changes. Duplicate changes can be seen as wasted resources that one needs to prevent. However, for cross-component changes, we observe that the most important category is for testing purposes, which one does not need to prevent. Instead, developers explicitly create a change that is meant to be abandoned since it is just for testing purposes. Furthermore, Wang et al. [29] found that the testing category accounts just for 4.39% of their studied changes. We also think that there is a lack of synchronization among different teams which leads to a Duplicate category of abandoned changes. As shown in the example of Table 6, a change was already implemented in a different project. Such wasted resources are at the level of different components, which can be more difficult to identify on a large scale of components rather than just already implemented changes within the same component.

Summary of RQ4

Based on our qualitative analysis, we identified a variety of reasons between a pair of cross-component dependent changes, among which the Configuration-related changes are the most frequent purpose (34.64%). Furthermore, we explored various categories for which OpenStack developers decide to abandon a given cross-component change. Thus, our findings indicate that changes that were abandoned for testing proposes are the most prevalent (38.45%), followed by Dupldiate (15.42%), and then Lack of activity (12.63%). As a result, **we recommend developers to take these findings into account when making a change across components in the future. We also recommend researchers design approaches to better handle cross-project changes**.

RQ5. Who is responsible for addressing the cross-project/service changes?

Motivation: The goal of this research question is to examine the resources in terms of developers and reviewers required for cross-project/service-dependent changes. A pair of two dependent changes made by different developers requires different team members to engage and discuss their changes, whereas two dependent changes in different components made by the same developer require that developer to be familiar with both projects. Thus, we determine whether dependent changes are developed by different developers. If not, whether a developer responsible for a cross-project dependency has the same maturity over both projects or whether they end up working on projects they are not familiar with. Similarly, we evaluate to which extent OpenStack reviewers need to collaborate with other team members to review changes that are cross-project/service-dependent.

Approach: To determine whether dependent changes are made by the same or different developers, we leverage dependencies involving different projects/services that are obtained following the approach discussed in the methodology section 3. We identify the developers of cross-project/service-dependent changes and quantify the number of dependent pairs that share the same developer vs. different developers.

For changes carried out by the same developer, we evaluate the expertise of that developer in both projects to see whether developers end up contributing to projects they are less familiar with. Particularly, we would like to know if a developer changing two projects, is the same developer an expert in both projects or if he/she is an expert in one project and end up working on a project he/she is less familiar with. To this end, we measure the developers' maturity in the involved projects as the sum of changed lines of code that a developer made in a project over the total number of changed lines of code in the same project (at the creation time of a studied cross-component change), according to the same approach used by Li et al. [49]. For example, for two dependent changes A and B that are made in different projects "X" and "Y" by the same developer. We count the number of changed lines of code I that the developer had in the "X" project before and including the change A and the number of changed lines of code J that he/she had in the "Y" project before and including the change B. After that, we identify the min and max values, as respectively being, the minimum and maximum number of changed lines of code a developer had in the pair of changes in a studied cross-project dependency. Hence, we compare the maturity of OpenStack developers on projects they are more familiar with over projects they are less familiar with of a dependency as $max(I, J)/min(I, J)$. Consequently, this allows us to know how much experience a developer has in the project with more familiarity compared to the experience he/she has in the project which is less experienced on. Note that we also report $max(I, J)$ and $min(I, J)$ separately to quantify the experience in both projects of a dependency.

Moreover, we study how different cross-project dependent changes are made by different developers vs. those made by the same developer in terms of the complexity of changes and review efforts which are measured using the three following metrics:

– **Code churn:** We use this metric similarly to a prior work [6], to better understand the number of changed lines of code of two dependent changes, and see how significant it is among those made by different developers and dependent changes made by the same developer. To do so, code churn is identified by the number of changed lines in a change (i.e., deleted and newly added lines of code).
– **Duration:** We also compare merge duration between dependent changes made by the same vs. different developers. We define the duration as the time between the creation of change and the time that change is merged.

– **Revision:** Similarly, we count and compare the number of revisions of each two dependent changes when they are developed by the same developer vs. different developers.

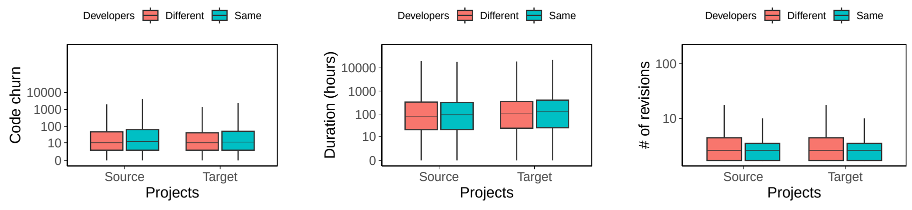
We leverage these previous metrics to compare pairs of dependent **C**hanges made by the **S**ame developers (CS_1, CS_2) vs. pairs of dependent **C**hanges made by **D**ifferent developers (CD_1, CD_2). We define CS_1 and CD_1 as the source of a dependency and CS_2, CD_2 as the target of a dependency. The target change is the one with the "Depends-On" tag, and the other change is the source when two dependent changes are linked with the "Depends-On" tag. For the "Needed-By" tag, we identify changes containing that tag as the source, whereas the other one is marked as the target. Although the "Related-Bug" and "Change-id" do not mention such meaningful relation between two changes in the same way as the two previous tags. Hence, we define the source as the change created first, while the change created later is considered the target.

Similarly, we assess the level of cooperation among OpenStack reviewers who address cross-project and cross-service dependencies. To accomplish this, we first identify the reviewers involved in each dependent change. Then, we compute the percentage of common reviewers among the studied cross-project/service-dependent changes for each pair of dependent changes. Note that we do not consider dependent changes that are published by user bots, and similarly to the previous RQ, we consider only merged changes.

Table 7: Number of cross-project/service dependencies addressed by same and different developers.

	Same developers	Different developers
Cross-project	35,556	11,572
Cross-service	5,911	2,944

Results: 24.55% and 33.24% of our studied cross-project and cross-service dependencies, respectively, were developed by different developers. Table 7 shows the number of cross-project/service dependencies that are made by the same vs. different developers. Although 24.55% of the studied cross-project dependencies were tackled by distinct developers, we observe that 71.63% of these dependencies have at least one developer participating in the review process of the second change, whereas both developers of 27.44% of the pairs of cross-project dependencies were involved in reviewing the other projects changes. For example, the dependent changes *841489* (in "Requirements" project) and *841712* (in "Neutron" project) were, respectively, developed by R. A. and S. K., whereas R. A. participated in the review of *841712* and S. K. in the review of *841489*. Similar observations were perceived for cross-service dependencies. By focusing on dependent changes whose developers participate in reviewing the other pair of projects, 77.06% of these dependent changes involve developers voting during the review process, while 21.12% include developers providing specific comments. However, we do not observe any case where a developer participates in the review of the second project by uploading a new version of a change (or revision). Note that there are also other kinds in which a reviewer contributes to the change review process. For example, we notice from the two dependent changes *831935* and *844680*, that the author of the *831935* change was added as a reviewer in *844680*.



(a) Code churn. (b) Duration. (c) # of revisions.

Fig. 21: Comparison of (a) **code churn**, (b) **duration**, and (c) **number of revisions** between same and different developers addressing “Source” (1st change) and “Target” (2nd change) of a dependency.

Although dependent changes made by the same or different developers have the same code churn, the 2nd change of a dependency is 1.14 (125.8/109.92) times slower when the pair of changes are made by the same developers. We observe that two cross-project dependent changes (“Source” and “Target”) that are made by the same developer have, respectively, a median code churn of 12 and 11, whereas dependent changes that are made by different developers have a median code churn 10 in both changes, as shown in Figure 21a. Furthermore, we notice that the median duration of a review change of two dependent changes made by the same developers is greater than that of the different developers, accounting for 93.05 and 125.8 hours, and 80.5 and 109.92 hours, respectively, as shown in Figure 21b. For example, the target (*531138*) of the following dependent changes (*531196* and *531138*), which are made by the same developer, was merged in 125.04 hours (5.2 days). Moreover, we observe that the median number of revisions worked out in the review process of the “Source” and “Target” changes are the same for the same and different developers, as shown in Figure 21c. Table 8 shows how such metrics are statistically significantly different from each other between the same and different developers.

Developers addressing both cross-project and cross-service dependent changes, have a median of three times and eight times more experience in one project over the other one at the time of the cross-project and cross-service change, respectively. According to Figure 22a, we notice that developers have substantial experience in one project compared

Table 8: Wilcoxon rank sum test for code churn, duration, and the number of revisions, wrt., source, and target. Note that significant $p-values$ (i.e., $p-value \ll 0.05$) are displayed in bold text format.

Metric	**Source**	**Target**
Code churn	$\mathbf{3.895 \times 10^{-6}}$	0.1948
Duration	0.6749	$\mathbf{6.177 \times 10^{-8}}$
# of revisions	$< \mathbf{2.2 \times 10^{-16}}$	$< \mathbf{2.2 \times 10^{-16}}$

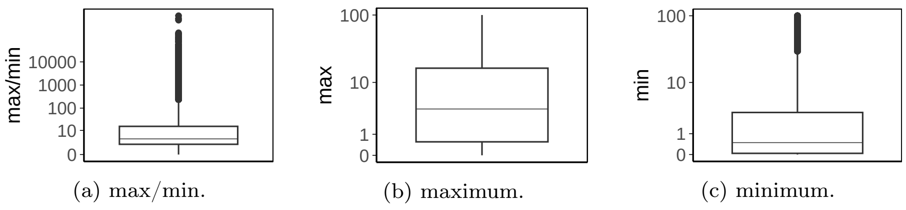
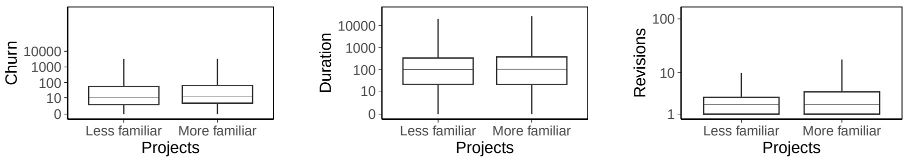


Fig. 22: Distribution of (a) **experience** of a developer denoted as the division of maximum and minimum values of a developer maturity in a pair of projects, (b) **maximum** and (c) **minimum** values of developer maturity in one project.

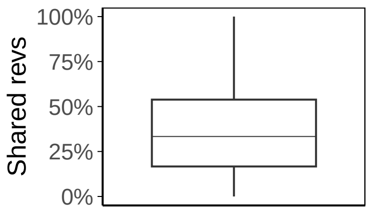
to the other one, suggesting that they end up working on a project they are not as familiar with as the first project. Developers can have a median of 3.58% experience in the project they are more familiar with, while 0.4805% for projects they are less familiar with, as shown in Figures 22b and 22c, respectively. Note that the observed differences (i.e., between min(I, J) and max(I, J)) are statistically significantly different (Wilcoxon rank sum test, $p\text{-}value = 2.2 \times 10^{-16} \ll 0.05$). Interestingly, 20.17% of studied cross-project dependencies reveal that such developers made no changes in at least one of the projects' pairs. A possible explanation for our observation is that developers are responsible for testing the impact of their changes on other components and fixing any issues in other components accordingly.



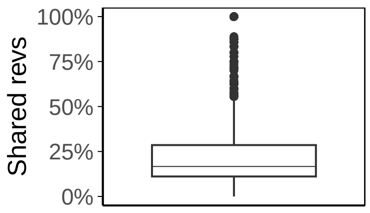
(a) Code churn of two dependent changes. (b) Duration of two dependent changes. (c) # of revisions of two dependent changes.

Fig. 23: (a) **code churn**, (b) **duration**, and (c) **number of revisions** of two dependent changes developed by the same developer broken down into less and more familiar projects.

The median code churn and duration are relatively higher for projects a developer is familiar with, while the median number of revisions is similar among less and more familiar projects. We observe that projects a developer has more experience highlight a median code churn of 13 compared to the one he/she is less experienced in (a median of 11), as shown in Figure 23a. Moreover, we notice, from Figure 23b, that it takes a median of 106.42 hours to get changes merged of the project a developer is familiar with compared to 101.2 hours to the project he/she is less familiar with. Wilcoxon rank test shows that the time it takes to get changes merged into the project with less familiarity vs. those with more familiarity is statistically significantly different (p-value = 2.2e-16 « 0.05). However, we observe that the median number of revisions (a median of 2) is equal across projects in which developers are less and more experienced, as depicted in Figure 23c.



(a) % of shared reviewers between cross-project dependencies.



(b) % of shared reviewers between cross-service dependencies.

Fig. 24: Percentage of shared reviewers between (a) **cross-project** and (b) **cross-service** dependencies.

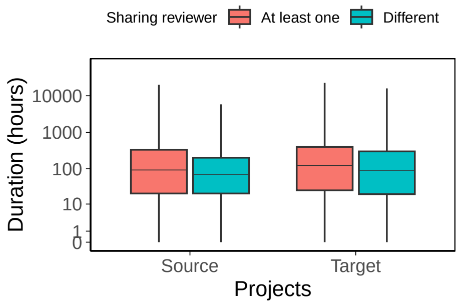


Fig. 25: Comparison of merge duration between two cross-project dependent changes sharing no reviewers and those sharing at least one reviewer.

8.65% and 12.03% of our investigated cross-project and cross-service dependencies, respectively, were not reviewed by any common reviewer. While the examined cross-project dependencies share a median of 33.33% of their respective reviewers, we observe that 8.65% of these dependencies did not reveal any common reviewer. Similarly for services, even though dependencies involving different services share a median of 16.66% of their corresponding reviewers, 12.03% of these dependencies are not carried out by any shared reviewer, as depicted in Figure 24. Furthermore, we observe that 49.05% of such cross-project dependencies sharing no reviewer were created in different releases. Interestingly, we observe that cross-project-dependent changes are quickly merged when they share no common reviewer vs. sharing developers. For instance, two dependent changes, sharing at least one reviewer, are merged after a median of 92.81 hours (source) and 123.83 hours (target); however, when two dependent changes do not share any common reviewer, they are merged after a median of 70.81 hours (source) and 91.39 hours (target) as shown in Figure 25. Additionally, we observe that the merge duration for dependent changes sharing vs. not sharing a reviewer is statistically significant (Wilcoxon rank sum test, source: $\textit{p-value} = 3.131 \times 10^{-14} \ll \textit{0.05}$; target: $\textit{p-value} = 2.2 \times 10^{-16} \ll \textit{0.05}$).

Summary of RQ5

While nearly a quarter of cross-project dependencies are made by different developers, these developers frequently take part in the review process of the second change. However, we notice that when developers are involved in both projects of a dependency, they tend to make more significant modifications in the project they are less experienced with, as opposed to the one they are familiar with. **Our results suggest that managers choose the right contributor for cross-project change as it can have an impact on the time to develop two dependent changes.**

5 Implications

In this section, we recommend a set of implications from our results to practitioners and researchers for better maintenance of multi-component systems.

5.1 Implications for Practitioners:

We recommend managers do not neglect in their planning and resource estimation that a change might require modifying multiple components that are managed by different teams and leverage our qualitative findings for hints on the types of changes that might involve multiple components. The changes that require modifying multiple components can be related to the complexity of the releases, the types of projects, and types of changes that we identified in our qualitative study.

In fact, we observe that different projects are managed by different teams, suggesting that these components are less likely to be maintained by different teams in an independent way (PQ). However, as we observe in RQ1, cross-component-changes represent a large number of changes and such a type of dependent change occurs in all of our studied releases as shown in RQ2. We also want to shed light that a change might trigger a whole chain of changes, while such a chain implies just two components, a chain can have a larger number of components (up to 224 components). At the same time, we observe that a good number of cross-project changes just end up being abandoned (RQ3). Thus, our results suggest managers not neglect the fact that a change might imply other changes in other components with a good chance (20.85%) might end up being abandoned.

We also want to shed light on the fact that the more complex a release is, the higher the chance for cross-project changes (RQ2). Leveraging our qualitatively obtained reasons for cross-component changes can also help practitioners identify changes that are more likely to trigger changes in other components. For example, upgrading the configuration is likely to introduce changes in other components. Thus, practitioners need to consider more resources to implement such a type of change. Hence, this is aligned with our prior work [7, 20] in which we found that cross-component configuration errors are common and difficult to identify.

Our results suggest managers consider that certain types of projects are more likely to be co-changed, hence planning more time and resources for the modification of these projects. For instance, we observe that a larger portion of cross-project changes is related to configuration (RQ4). Besides, even after refactoring OpenStack by archiving multiple projects, certain projects were more likely to be part of cross-project changes such as openstack-zuul-jobs and project-config (RQ2), which are for the DevOps pipeline and configuration. In fact, it is interesting to note that OpenStack tried to have ownership over its dependencies by moving the declaration of dependencies to the Requirement project, while such a move created more cross-project changes as an upgrade to a dependency needs to go through that requirement project. Each upgrade might also create maintenance overhead in other components and increase the chance of cross-project changes (as shown in the large example of cross-project changes in RQ1). We suggest practitioners follow our mining software repositories approach to identify the components that are more likely to be involved in cross-component changes and especially suggest practitioners leverage tags that indicate cross-component dependent changes such as the "depends-on" tag.

To minimize the amount of cross-component changes, we suggest practitioners refactor the architecture of their system even when it is a multi-component system. For instance, having a modular architecture might not be a lifelong solution, as such we recommend practitioners consider restructuring their components with a large number of cross-project changes. For instance, we observe that the number of cross-component changes was increasing over different releases to reach a peak at the version Rocky and

Stein, while such a number continuously decreased after archiving a large number of projects in the Victoria release. That large refactoring of projects also reduced the number of projects a given project depends on.

Our results suggest practitioners choose the right resources to develop cross-project dependent changes. In fact, we observe that when two dependent changes are made by the same developer, that developer ends up working on a project that he/she is less familiar with. As RQ5 indicates, a developer is three times more experienced in one project (less familiarity) than the other one (more familiarity). When the pair of dependent changes are made by different developers, the developer of the first change participates in the review process of the second change. We also recommend managers consider the same developer for both changes when possible, as dependent changes made by the same developers are more likely to be quickly merged compared to dependent changes made by different developers (RQ5).

5.2 Implications for Researchers:

Our results suggest future studies recommend components that are likely to co-change with a given component. As we found in RQ1 and RQ2, the cross-component changes are prevalent and increasingly evolving over time, whereas each of the components is maintained by a different team, as confirmed by our PQ. To assist practitioners in the planning of their changes, we suggest future studies develop approaches that leverage different techniques, such as machine learning and different metrics, including the purpose of changes to predict what other components one needs to change.

Our results suggest researchers develop solutions that recommend which other components to test given one component change. For instance, we observe in RQ2 that a good number of abandoned cross-project changes are meant to be abandoned as they were for testing the impact of a change in a given component on other components. We also observe that the majority (38.45%) of qualitatively analyzed abandoned cross-project changes are related to the Test category. Thus, we recommend researchers propose models to predict what components to test when changing a given component. We believe that such recommendation systems can also help the integration and deployment pipeline, as instead of testing the whole system, one might prioritize testing components that are more likely to be impacted by changing a given component.

Our results suggest researchers develop approaches that recommend when an architecture refactoring is required to minimize the cross-component maintenance dependencies. In fact, as we observe that the number of co-changes increases up to the time when OpenStack archived a large number of projects, we recommend future studies to develop approaches that can suggest when a project needs to be restructured. For instance, most of the studies (as discussed in related work 2) focus on migrating a monolithic system to microservices, while migration can still occur afterward. In practice, developers should be notified when there is an unexpected number of co-changes involving a given project. A large pool of prior work proposed various coupling measurements between components of a software system including microservice-level coupling [48] and organizational coupling [49]. Such a coupling metric can be used as an indicator for refactoring needs.

Our results suggest researchers investigate the impact of different components' refactoring strategies on the co-evolution of the components of a multi-component system. We find that the number of cross-component changes increases over different releases up to a certain time in which OpenStack archived a large number of projects. From then, the number of cross-component changes decreased over time. While we cannot directly conclude that reducing the number of components will impact the number of cross-component changes, we suggest future studies to investigate the impact of different projects' refactoring techniques on the number of cross-component changes. Among the refactoring techniques we suggest is the removal of small projects and the combination of similar projects or projects that frequently co-change. One way to achieve this is to leverage different clustering algorithms such as KNN and determine how related projects can be combined according to their characteristics, which we plan to investigate in future work.

Our results suggest researchers extend existing approaches related to wasting efforts due to code duplication or abandoned changes to the level of cross-component systems changes. Similarly to Wang et al. [29], we observe that duplicate changes are a common reason for cross-component abandoned changes (15.42%). Yet, we observe cases in which the duplication itself is cross-component. For example, we observe cases in which a change was abandoned since it was already implemented in another component. Hordijk et al. [2] stated that source code duplication is a major factor impacting software quality. Hence, we suggest future studies on code duplication to extend their work to the cross-component level. We also suggest future studies to re-evaluate the model of Khatoonabadi et al. [46] in the identification of cross-component abandoned changes and evaluate whether the model is as performed on these cross-component changes compared to single-component duplicated changes.

6 Threats to Validity

6.1 External Validity

Our study and a large number of previous studies [37, 39, 45, 47] that focused on OpenStack share the same threat to validity related to the generalizability of our results to other multi-component software systems. We focus on OpenStack since it is a well-known project, studied by a large body of research, has a large number of projects that better represent complex multi-component systems, and especially has a well-defined way of declaring dependencies between changes that allows us to systematically identify cross-project and cross-service changes. We also recommend future work and practitioners replicate our study on their multi-component systems to better understand the co-evolution of their components and better manage such an evolution.

6.2 Internal Validity

An internal threat to validity is related to our manual analysis. Our manual analysis can be biased with the selected sample of cross-project/service changes. To mitigate such a risk, we select a random sample. Also, studying more cross-project changes can reveal more types of cross-project changes. We encourage future studies to replicate our manual analysis on another sample and other projects to identify more possible reasons for cross-project/service-dependent changes.

7 Conclusion

Modern software systems are composed of multiple components (aka., multi-component systems) that are expected to evolve independently from each other for better flexibility and agility. However, previous studies found that reaching a completely independent evolution is difficult to achieve. Yet, we observe from our empirical study on OpenStack that even if the components are maintained by different teams (i.e., the concept of ownership), these components depend on each other, leading to cross-project and even cross-service-dependent changes. Such dependent changes are common, as we observe 52,069 cross-project changes in OpenStack. We also observe that such a number is continuously evolving up to a certain point in which OpenStack decided to archive a large number of projects. With that, the number of cross-project-dependent changes is still relevant. We quantitatively and qualitatively observe from our study that these dependent changes are for different reasons, including configuration-related changes, moving code to the right location, and testing the impact of a change on other components, whereas developers often create a change for testing purposes on other components then abandon such a change. Furthermore, we observe that both changes of a pair of cross-project changes can be made by the same developer, who ends up working on a project she or he is less familiar with, or the two different changes are by different developers, yet they collaborate on reviewing the second change. Our study has several implications for practitioners and researchers. Among these implications is that managers should always pay attention to cross-project-dependent changes in their planning, and our qualitative analysis provides a hint into when such a cross-project or service change might occur, and when a cross-component change might end up being abandoned Our results shed light on the importance of restructuring a project to minimize the dependency between different components. For researchers, we recommend building approaches that predict co-changes by taking into account the possible reasons for such change dependencies, when an architecture needs to be restructured, what components need to be tested given a change in a component and recommend the appropriate developer and reviewer for cross-project changes.

Declaration of Conflict of Interest

The authors declared that they have no conflict of interest in the submission of this manuscript.

Data Availability Statements

The location of the data and the related code used to carry out this study is provided in the following GitHub repository: https://github.com/aliarabat/OpenStack/tree/stable.

References

[1] Kyle Gabhart and Bibhas Bhattacharya. *Service Oriented Architecture Field Guide for Executives*. Wiley Publishing, 2008. ISBN: 0470260912.

[2] Wiebe Hordijk, María Laura Ponisio, and Roel Wieringa. "Harmfulness of Code Duplication - A Structured Review of the Evidence". In: *Journal of Signal Processing Systems* (2009). URL: https://api.semanticscholar.org/CorpusID:17418919.

[3] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009. ISBN: 9781457102196.

[4] Steven Raemaekers, Arie van Deursen, and Joost Visser. "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository". In: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 2014, pp. 215–224. DOI: 10.1109/SCAM.2014.30.

[5] Kamil Jezek, Jens Dietrich, and Premek Brada. "How Java APIs Break - An Empirical Study". In: *Inf. Softw. Technol.* 65.C (Sept. 2015), pp. 129–146. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2015.02.014.

[6] Yujuan Jiang and Bram Adams. "Co-Evolution of Infrastructure and Source Code: An Empirical Study". In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. MSR '15. Florence, Italy: IEEE Press, 2015, pp. 45–55. ISBN: 9780769555942. DOI: 10.1109/MSR.2015.12.

[7] Mohammed Sayagh and Bram Adams. "Multi-layer software configuration: Empirical study on wordpress". In: *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2015, pp. 31–40. DOI: 10.1109/SCAM.2015.7335399.

[8] Jia Tong et al. “Characterizing and Predicting Bug Assignment in OpenStack”. In: *2015 Second International Conference on Trustworthy Systems and Their Applications*. 2015, pp. 16–23. DOI: 10.1109/TSA.2015.14.

[9] Yoji Yamato et al. “Software Maintenance Evaluation of Agile Software Development Method Based on OpenStack”. In: *IEICE Transactions on Information and Systems* 98.7 (2015), pp. 1377–1380. DOI: 10.1587/transinf.2015EDL8049.

[10] Ivan Candela et al. “Using Cohesion and Coupling for Software Remodularization: Is It Enough?” In: *ACM Trans. Softw. Eng. Methodol.* 25.3 (June 2016). ISSN: 1049-331X. DOI: 10.1145/2928268.

[11] Susan J. Fowler. *Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization*. 1st. O’Reilly Media, Inc., 2016. ISBN: 1491965975.

[12] Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa. “More Common Than You Think: An In-depth Study of Casual Contributors”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. 2016, pp. 112–123. DOI: 10.1109/SANER.2016.68.

[13] Amazon Web Services. *Introduction to Microservices*. Sept. 2016. URL: https://www.slideshare.net/AmazonWebServices/introduction-to-microservices-66320469 (visited on 06/23/2023).

[14] Chen-Yuan Fan and Shang-Pin Ma. “Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report”. In: *2017 IEEE International Conference on AI & Mobile Services (AIMS)*. 2017, pp. 109–112. DOI: 10.1109/AIMS.2017.23.

[15] Jean-Philippe Gouigoux and Dalila Tamzalit. “From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture”. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 62–65. DOI: 10.1109/ICSAW.2017.35.

[16] Raula Gaikovina Kula et al. “On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem”. In: *CoRR* abs/1709.04638 (2017). arXiv: 1709.04638.

[17] Genc Mazlami, Jürgen Cito, and Philipp Leitner. “Extraction of Microservices from Monolithic Software Architectures”. In: *2017 IEEE International Conference on Web Services (ICWS)*. 2017, pp. 524–531. DOI: 10.1109/ICWS.2017.61.

[18] Tomonobu Niwa, Yuki Kasuya, and Takeshi Kitahara. “Anomaly detection for openstack services with process-related topological analysis”. In: *2017 13th International Conference on Network and Service Management (CNSM)*. 2017, pp. 1–5. DOI: 10.23919/CNSM.2017.8255977.

[19] Adalberto R. Sampaio et al. “Supporting Microservice Evolution”. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017, pp. 539–543. DOI: 10.1109/ICSME.2017.63.

[20] Mohammed Sayagh, Noureddine Kerzazi, and Bram Adams. "On Cross-Stack Configuration Errors". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017, pp. 255–265. DOI: 10.1109/ICSE.2017.31.

[21] Manel Abdellatif et al. "State of the Practice in Service Identification for SOA Migration in Industry". In: *Service-Oriented Computing*. Ed. by Claus Pahl et al. Cham: Springer International Publishing, 2018, pp. 634–650. ISBN: 978-3-030-03596-9. DOI: 10.1007/978-3-030-03596-9_46.

[22] Andrei Furda et al. "Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency". In: *IEEE Software* 35.3 (2018), pp. 63–72. DOI: 10.1109/MS.2017.440134612.

[23] C. Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018. ISBN: 9781617294549. URL: https://books.google.com/books?id=UeK1swEACAAJ.

[24] Justus Bogner et al. "Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 546–556. DOI: 10.1109/ICSME.2019.00089.

[25] Lorenzo De Lauretis. "From Monolithic Architecture to Microservices Architecture". In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2019, pp. 93–96. DOI: 10.1109/ISSREW.2019.00050.

[26] Jonas Fritzsch et al. "Microservices Migration in Industry: Intentions, Strategies, and Challenges". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 481–490. DOI: 10.1109/ICSME.2019.00081.

[27] Bruno L. Sousa, Mariza A. S. Bigonha, and Kecia A. M. Ferreira. "Analysis of Coupling Evolution on Open Source Systems". In: *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*. SBCARS '19. Salvador, Brazil: Association for Computing Machinery, 2019, pp. 23–32. ISBN: 9781450376372. DOI: 10.1145/3357141.3357147.

[28] José Apolinário Teixeira and Helena Karsten. "Managing to release early, often and on time in the OpenStack software ecosystem". In: *Journal of Internet Services and Applications* 10.1 (Dec. 2019), p. 32. ISSN: 1867-4828. DOI: 10.1186/s13174-019-0105-z.

[29] Qingye Wang et al. "Why is my code change abandoned?" In: *Information and Software Technology* 110 (2019), pp. 108–120. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2019.02.007.

[30] Wei Zheng et al. "Towards understanding bugs in an open source cloud management stack: An empirical study of OpenStack software bugs". In: *Journal of Systems and Software* 151 (2019), pp. 210–223. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.02.025.

[31] Manel Abdellatif et al. "A Type-Sensitive Service Identification Approach for Legacy-to-SOA Migration". In: *Service-Oriented Computing:*

18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings. Dubai, United Arab Emirates: Springer-Verlag, 2020, pp. 476–491. ISBN: 978-3-030-65309-5. DOI: 10.1007/978-3-030-65310-1_34.

[32] Chia-Yu Li, Shang-Pin Ma, and Tsung-Wen Lu. "Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System". In: *2020 International Computer Symposium (ICS)*. 2020, pp. 519–524. DOI: 10.1109/ICS51289.2020.00107.

[33] Jakša Vučković. "You Are Not Netflix". In: *Microservices: Science and Engineering*. Ed. by Antonio Bucchiarone et al. Cham: Springer International Publishing, 2020, pp. 333–346. ISBN: 978-3-030-31646-4. DOI: 10.1007/978-3-030-31646-4_13.

[34] Manel Abdellatif et al. "A taxonomy of service identification approaches for legacy software systems modernization". In: *Journal of Systems and Software* 173 (2021), p. 110868. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110868.

[35] Filipe Roseiro Cogo, Gustavo A. Oliva, and Ahmed E. Hassan. "An Empirical Study of Dependency Downgrades in the npm Ecosystem". In: *IEEE Transactions on Software Engineering* 47.11 (2021), pp. 2457–2470. DOI: 10.1109/TSE.2019.2952130.

[36] César Soto-Valero et al. "A comprehensive study of bloated dependencies in the Maven ecosystem". In: *Empirical Software Engineering* 26.3 (Mar. 2021), p. 45. ISSN: 1573-7616. DOI: 10.1007/s10664-020-09914-8.

[37] Eman Abdullah AlOmar et al. "Code Review Practices for Refactoring Changes: An Empirical Study on OpenStack". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. MSR '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 689–701. ISBN: 9781450393034. DOI: 10.1145/3524842.3527932.

[38] Md Atique Reza Chowdhury et al. "On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages". In: *IEEE Transactions on Software Engineering* 48.8 (2022), pp. 2695–2708. DOI: 10.1109/TSE.2021.3068901.

[39] Armstrong Foundjem et al. "A mixed-methods analysis of micro-collaborative coding practices in OpenStack". In: *Empirical Software Engineering* 27.5 (June 2022), p. 120. ISSN: 1573-7616. DOI: 10.1007/s10664-022-10167-w.

[40] *OpenStack is dead? The numbers speak for themselves*. Mar. 2022. URL: https://ubuntu.com/blog/openstack-is-dead (visited on 04/25/2023).

[41] Imen Trabelsi et al. "From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis". In: *Journal of Software: Evolution and Process* (2022), e2503. DOI: 10.1002/smr.2503.

[42] Yalemisew Abgaz et al. "Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review". In: *IEEE Transactions on Software Engineering* (2023), pp. 1–32. DOI: 10.1109/TSE.2023.3287297.

[43] Wesley K.G. Assunção et al. “How do microservices evolve? An empirical analysis of changes in open-source microservice repositories”. In: *Journal of Systems and Software* 204 (2023), p. 111788. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2023.111788.

[44] Mohamed Amine Batoun et al. “An Empirical Study on GitHub Pull Requests’ Reactions”. In: *ACM Trans. Softw. Eng. Methodol.* 32.6 (Sept. 2023). ISSN: 1049-331X. DOI: 10.1145/3597208.

[45] Narjes Bessghaier et al. “What Constitutes the Deployment and Runtime Configuration System? An Empirical Study on OpenStack Projects”. In: *ACM Trans. Softw. Eng. Methodol.* 33.1 (Nov. 2023). ISSN: 1049-331X. DOI: 10.1145/3607186.

[46] Sayedhassan Khatoonabadi et al. “On Wasted Contributions: Understanding the Dynamics of Contributor-Abandoned Pull Requests–A Mixed-Methods Study of 10 Large Open-Source Projects”. In: *ACM Trans. Softw. Eng. Methodol.* 32.1 (Feb. 2023). ISSN: 1049-331X. DOI: 10.1145/3530785.

[47] Dong Wang et al. “An Exploration of Cross-Patch Collaborations via Patch Linkage in OpenStack”. In: *IEICE Transactions on Information and Systems* E106.D (Feb. 2023), pp. 148–156. DOI: 10.1587/transinf.2022MPP0002.

[48] Chenxing Zhong et al. “On measuring coupling between microservices”. In: *Journal of Systems and Software* 200 (2023), p. 111670. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2023.111670.

[49] Xiaozhou Li, Dario Amoroso d’Aragona, and Davide Taibi. “Evaluating Microservice Organizational Coupling Based on Cross-Service Contribution”. In: *Product-Focused Software Process Improvement.* Ed. by Regine Kadgien et al. Cham: Springer Nature Switzerland, 2024, pp. 435–450. ISBN: 978-3-031-49266-2.

[50] *Statistics about OpenStack.* URL: https://openinfra.dev (visited on 04/24/2023).