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

Release Synchronization in Software Ecosystems
Empirical Study on OpenStack

Armstrong Foundjem · **Bram Adams**

Received: date / Accepted: date

Abstract Software ecosystems bring value by integrating software projects related to a given domain, such as Linux distributions integrating upstream open-source projects or the Android ecosystem for mobile Apps. Since each project within an ecosystem may potentially have its release cycle and roadmap, this creates an enormous burden for users who must expend the effort to identify and install compatible project releases from the ecosystem manually. Thus, many ecosystems, such as the Linux distributions, take it upon them to release a polished, well-integrated product to the end-user.

However, the body of knowledge lacks empirical evidence about the coordination and synchronization efforts needed at the ecosystem level to ensure such federated releases. This paper empirically studies the strategies used to synchronize releases of ecosystem projects in the context of the OpenStack ecosystem, in which a central release team manages the six-month release cycle of the overall OpenStack ecosystem product.

We use qualitative analysis on the release team's IRC-meeting logs that comprise two OpenStack releases (one-year long). Thus, we identified, cataloged, and documented ten major release synchronization activities, which we further validated through interviews with eight active OpenStack senior practitioners (members of either the release team or project teams). Our results suggest that even though an ecosystem's power lies in the interaction of inter-dependent projects, release synchronization remains a challenge for both the release team and the project teams. Moreover, we found evidence (and reasons) of multiple release strategies co-existing within a complex ecosystem.

1 Introduction

A software ecosystem comprises a set of socio-technically inter-independent software projects on top of a given technological platform [1, 2, 3]. The last decade has seen a proliferation

Armstrong Foundjem
Queen's University, Kingston
E-mail: a.foundjem@queensu.ca

Bram Adams
Queen's University, Kingston
E-mail: bram.adams@queensu.ca

of such software ecosystems [4, 5, 6], from programming language-related ecosystems like Maven and npm to operating system distributions like Debian and Fedora, infrastructure-related ecosystems like OpenStack and Eclipse, and mobile app ecosystems like Android and *iOS*. While each project within an ecosystem is managed locally by its project teams (or organization), the strength of an ecosystem lies in both the social interactions between these project/cross-project teams [7, 8] and their reuse of functionality via technical dependencies [9, 10, 11, 12]. Since each ecosystem project is autonomous, it will have its release cycle and roadmap to release new versions.

While this sounds ideal for programming language ecosystems like npm or Maven, or mobile app ecosystems like Android and *iOS*, independent roadmaps make it more challenging in practice for dependent projects to plan on when to update the new release of their dependency [13, 14, 15]. Worse, in distribution and infrastructure ecosystems, end-users are not interested in installing one project at a time but prefer to install polished and compatible versions of the most popular projects that form the core of the ecosystem. Afterward, end-users can complement this installation with individually picked releases of other ecosystem projects. The presence of purely asynchronous releases hampers this workflow.

Given this negative impact of asynchronous releases on end-users, many software ecosystems have developed ecosystem-level release synchronization strategies. Indeed, ecosystem release synchronization enables software projects following their roadmaps (release strategies) to engage in a standard and well polished (synchronized) deliverable at the end of the ecosystem release cycle. Usually, the central release team, whose mandate is to synchronize these individual project teams' release strategies, facilitates this engagement.

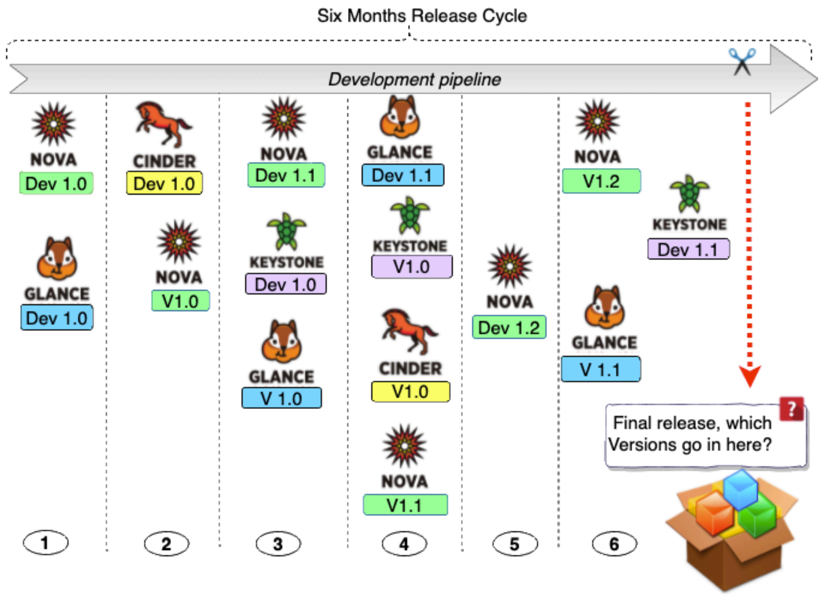
For example, Figure 1 shows the challenges that a typical release team faces each release cycle to *synchronize* the ecosystem releases. Let us suppose that Nova, at the beginning of a given release cycle, starts developing version Dev1.0 of an artifact and releases it in version v1.0, and continuously releasing different versions throughout the release cycle. Similarly, other projects such as Glance, Cinder, or Horizon follow different release roadmaps and have different versions of their releases. These different versions of projects' intermediary releases, for example, might have dependency constraints, which are not compatible with each other during ecosystem-level release. Now, at the end of the release cycle, which version of each given project will be available (or shipped) to the end-users?

In such a complex ecosystem, the centralized release team comes in to synchronize – package versions of artifacts that are compatible – and ship a well-polished and finished product of the ecosystem to the end-users. For example, in June 2006[1], Eclipse started rolling out "simultaneous" releases of the core Eclipse platform. Each simultaneous release packaged a set of essential and compatible plugins, removing the need for users to install the core platform and then manually install their plugins of choice. Similarly, OpenStack, since its third release in 2011 (Cactus) as shown in Figure 3, adopted a "federated" release synchronization strategy which is managed by a **centralized release team** instead of requiring projects to discuss conflicts peer-to-peer (as Eclipse requires).

While research on release engineering has exploded during the last five years [16, 17, 18, 19], most of this work focuses on the release process of individual projects, not ecosystems, especially in the context of rapid releases. Other researchers such as Adams et al. [20] studied the integration activities done by maintainers of individual packages in open source distributions between successive releases (e.g., updating to a new release or locally patching an identified bug). Similarly, Nayebi et al. [21] studied release strategies of individual mo-

[1] `shorturl.at/imFM1`

Fig. 1 Ecosystem release dependencies across project teams following different/multiple release models. Dev X .X means a given project team at some point in the six-month cycle is developing a milestone/intermediate release. V X.X implies that a project team released a specific version of a deliverable/milestone, etc., at some within the cycle.



bile apps. However, none of these studies consider the synchronization of releases between multiple ecosystem projects (packages)

Our work in this paper is motivated by the work of Teixeira et al. [22]. In their study, the authors synthesized online web resources to document the OpenStack ecosystem's release process and infrastructure and explain the different release strategies in the OpenStack ecosystem. Teixeira et al. reported the "what" about the ecosystem's rapid release process of the OpenStack ecosystem, but not the "how". For example, they found evidence of the different release strategies that co-exist in a complex ecosystem but did not determine why and how multiple release strategies exist in complex ecosystems such as OpenStack. Moreover, Teixeira et al. did not consider the release synchronization activities and challenges that the release team experienced at OpenStack. The authors encouraged future research to analyze the socio-technical activities to understand better the release process, especially the management of release notes.

Thus, in this paper, we perform an empirical study on the OpenStack open-source ecosystem to identify the "how" and "why" of the release synchronization process at the ecosystem level, specifically, what does it take for an ecosystem release team to manage synchronized releases across time successfully? We identify and mine the weekly Internet Relay Channel **(IRC**) communication logs of the OpenStack's release team to understand the activities that the release team performs weekly to coordinate and synchronize the ecosystem releases. IRC is an established, popular protocol for online chat and (textual) meetings between geographically distributed teams and is very commonplace in Open-source communities [23, 24]. We also classify the different release strategies supported by OpenStack, with concrete numbers of projects implementing those strategies for their deliverables and why such multiple release synchronization strategies co-exist in a complex ecosystem. Furthermore, we provide tangible evidence of ten socio-technical activities engaged by the release team during the coordination of project/cross-projects teams.

Furthermore, to validate our findings on the release synchronization activities, which we found, we interviewed eight active OpenStack senior practitioners (members of the release team and project teams) for correctness and completeness. In terms of accuracy, we were 100% correct and, however, for completeness, we missed one activity.

The main contributions of this research are as follows:

- **Qualitative analysis** of 52-weekly IRC online meeting logs, two online release plan documents, and two online release tracking documents spread across two OpenStack releases.
- **Identification** of 10 major activities involved in synchronizing releases between ecosystem projects/cross-project.
- **Validation** of the activities with four release team members and four OpenStack project members.
- **An empirical evidence** of why multiple release strategies co-exist within a complex ecosystem.
- A **generalization of our findings** on two major open-source communities.
- The **implications** of our findings to academics and practitioners.

2 Methodology

2.1 Subject System Selection

To empirically study release synchronization in software ecosystems, we looked for an ecosystem with the following characteristics:

- sufficient history of release synchronization
- well-documented release process
- open-source, allowing access to any data related to the actual releases as well as the process used
- manage multiple release strategies co-existing during release cycles
- archived communications about the majority of the release synchronization activities and plans

We carefully investigated the online documentation and archived release history of several open-source communities such as GNOME, Apache (ASF), Eclipse, OpenStack, Linux kernel, and the Kubernetes community. Out of the possible ecosystems satisfying the five criteria, we eventually selected the OpenStack ecosystem[2]. First of all, OpenStack is an open-source software ecosystem (under the Apache 2.0 license) for cloud computing developed by NASA and Rackspace in 2010. Besides, the first author is an OpenStack foundation member, making access to prominent individuals for interviews much easier

OpenStack, according to Forbes, is the "de facto standard for open-source based private clouds" [25], and is built on a principle of four opens: **Open** Source, **Open** Design, **Open** Development, and **Open** Community. Since its first ecosystem release, Austin[3], which consisted of only two projects, Nova and Cinder, the OpenStack ecosystem has seen a steady

[2]`https://www.openstack.org`

[3]the release naming convention at OpenStack follows an alphabetical order with each release's first letter. For example, **A**ustin, **B**exar, **C**actus, . . . , and **V**ictoria (current ecosystem release)

growth and now comprises of 759[4] projects (derived from 63 core projects)[5] [6] and involved 693 different companies. The ecosystem has over 20M lines of code contributed by over 100k community members, including volunteers, across 200 countries.

2.2 Communication Channel Selection

Among the different communication channels that exist in open source ecosystems and developers communities, such as the mailing list, IRC, Slack, Matrix, etc., OpenStack release team uses the IRC channel to discuss and coordinate projects/cross-project teams.

Specific mailing lists (openstack-discuss, release-announce and release-job-failures) and IRC channels (#openstack-release) are used by the release team. **openstack-discuss** is used for discussions related to OpenStack users and developers community, **release-announce** to announce OpenStack releases to the community, and **release-job-failures** to inform the release team about build failures in Zull (OpenStack's CI/CD pipeline).

All the discussions in these mailing lists related to release coordination and management are discussed in the weekly IRC meeting (on **#openstack-release**), with links to the message (threads). Similarly, any source of information, such as URLs (links), Wiki page, open review, or issues, is referenced in the IRC channel and is useful to coordinate projects/cross-project teams. Thus, members can follow those links and discuss the content in the IRC weekly release team meeting. For example, during a weekly meeting held on June 5th, 2015[7], during the Liberty development cycle, release team members were discussing a cross-project checklist for the Liberty milestone-1 and how to organize office hours. A release team member suggested that "maybe I should send an email to a list that will serve as a reminder that those things exist". The email was sent to cross-project teams and reference links to a Wiki page mentioned in the IRC channel that points to the announcement in the email.

In another example, discussions on the IRC channel on March 16th, 2018[8], during the Rocky development cycle, shows the conversation (and links to external URLs) among members of the release team on topics such as the release management onboarding process, a conceptual design for the release pipeline proposed by a team member, and many more.

We can see that the release team's IRC channel captures and logs essential information that facilitates release coordination. Therefore, the release team IRC-chat logs with all its content (including external links) is sufficient to understand the release management process at OpenStack.

Before the Mitaka OpenStack release, the release team used the #openstack-relmgr-office IRC channel to sync up with Project Team Leads (PTLs) and project team liaisons during the weekly project meeting and during release team office hours. However, this IRC channel became **#openstack-release**, the renaming ensured that all subscribers to the previous IRC channel had a 'bouncer' configured to continue communication uninterrupted in the new release channel (#openstack-release).

Introduce the 'openstack/release' repository, refining release models, and announcing measures to automate the release process.

[4] `shorturl.at/sADJ7`

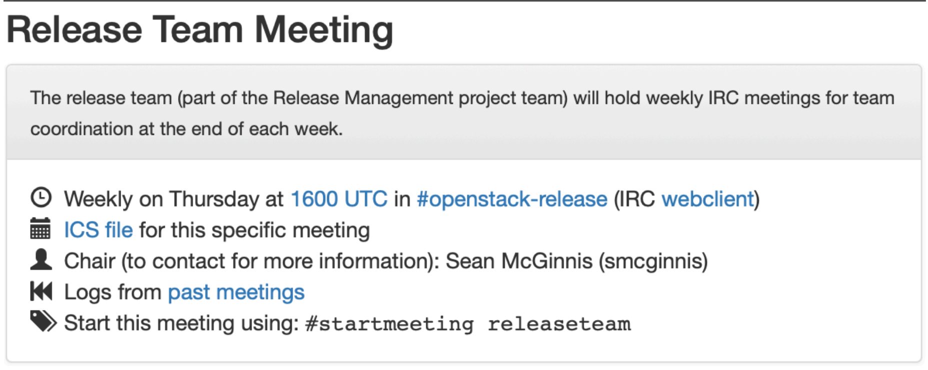
[5] `shorturl.at/moNQ6`

[6] For example, Nova is a core project, while $nova - powervm$ is a project (module), specifically, a sub-project of Nova with a separate development team.

[7] `releaseteam/2015/releaseteam.2015-06-05-13.01.log.html`

[8] `releaseteam/2018/releaseteam.2018-03-16-15.01.log.html`

Fig. 2 The release team mainly uses the IRC channel to coordinate weekly meetings.(http://eavesdrop.openstack.org/#Release_Team_Meeting), all past meeting logs and any change of the meeting time/date is available on the website, and publicly accessible.

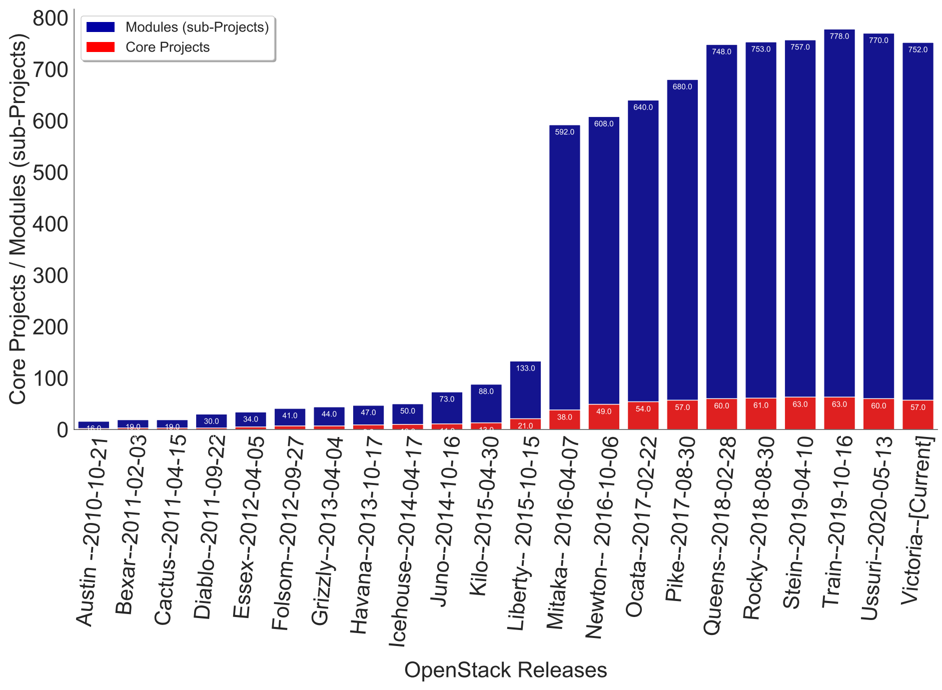


Release Team Meeting

The release team (part of the Release Management project team) will hold weekly IRC meetings for team coordination at the end of each week.

- Weekly on Thursday at 1600 UTC in #openstack-release (IRC webclient)
- ICS file for this specific meeting
- Chair (to contact for more information): Sean McGinnis (smcginnis)
- Logs from past meetings
- Start this meeting using: `#startmeeting releaseteam`

Fig. 3 OpenStack releases over time with different release cycles, 3-months cycle starting from Austin to Cactus and 6-months cycle starting from Diablo to current release.



2.3 Data Extraction from IRC Logs and Etherpad

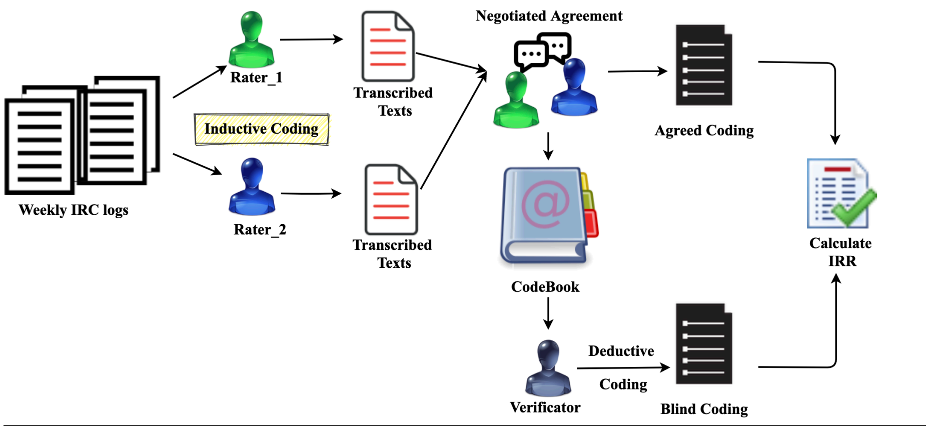
Given that the archives of IRC meeting notes of the release team[9] (see example in Figure 2) are only available from June 2015, we decide to study the activities of the release team within that period. We select two releases: Mitaka, the first release with complete meeting notes (from 10/2015 — 4/2016), and Queens, the most recent release at the time of this study with complete meeting notes (from 09/2017 — 03/2018). We considered several factors to determine the choice for an older and a more recent release. First of all, it enabled us to compare differences in activity distribution across time. Moreover, from Figure 3, we observe that the number of OpenStack core projects and associated projects (modules) exploded from the Mitaka release onward and started stabilizing by the Queens release.

Since each release follows a six-month cycle (26 weeks), we downloaded a total of 52 weekly IRC logs for the two studied releases. The logs for each hour-long weekly meeting is available in textual format (*.txt*, *.html*, and *.log*). Each file is named based on proximity to the next release and discusses the progress towards realizing the release plan. Since the OpenStack release cycle has 26 weeks of release team meetings on average, the meeting names range from R-25 (first meeting of the release-cycle), . . . , R-1, down to R+0, which is the release week.

Besides, we also downloaded the Etherpad documents containing the release plan's status and tracking at the time of each IRC meeting. The release plan includes the ongoing and open issues for the upcoming release, and it ultimately (during meeting R+0) is used to check whether the release has implemented all essential requirements. The tracking Etherpad is available online[10]. Based on each meeting's date and name, we downloaded the release plan at the time of each meeting.

2.4 Qualitative Analysis of IRC Logs and Etherpad

Fig. 4 We use inductive coding to build the codebook and deductive coding to compute the IRR using Cohen's Kappa for categorical data.



[9] `shorturl.at/jCEGI`
[10] `shorturl.at/kzEI7`

To identify the release synchronization activities performed by the release team, we use two rounds of open coding (inductive and deductive coding), as shown in Figure 4. Open coding aims at assigning relevant labels (tags) to chunks of texts (either at the phrase, sentence, or paragraph levels) within the weekly meeting logs and Etherpad documents [26, 27]. In qualitative analysis, these labels or tags are commonly called "codes;" (coding).

For example, in Figure 5, we show three different tags (Label #1, Label #2, and Label #3) that we placed on the highlighted text area. Next, we assigned labels to a code category (a code category represent a group of similar themes) to the bottom right. Initially, we assign these codes to a low-level (more specific) code category. Then as the coding continues, similar emerging low-level categories are clustered into a common theme, which we abstract as high-level code categories.

Inductive Coding. We performed two rounds of inductive coding. In these rounds, we aimed to generate new themes emerging from the data (IRC logs) to build a codebook [28, 29]. Both authors perform this round to calibrate the open coding to obtain high inter-rater reliability [30, 31, 32].

In particular, for the first round of inductive coding, both authors independently coded 15% of the weekly meetings [33, 34, 35] (the first eight of the Mitaka release) to identify different responsibilities of the release team members. For each discussion and agenda point, we reflected on the activity related to, for example, reviewing fixes for a showstopper bug or discussing the design of a new release tool. We assigned a code for each discovered activity and then annotated the corresponding discussion or agenda point with this code. During this process, we identified and recorded the hierarchical relations between codes. We used the online Dedoose [36] tool to annotate the logs and manage the codes, as shown in Figure 5.

After completing the first round of inductive coding, both authors discussed the resulting codes. The negotiated agreement aims to check for each annotated log excerpt whether its associated code makes sense, merging related codes into one or sub-code of a higher-level code. Once both authors consolidated their coding results, they both decided on the high-level code categories.

The first author continued the process and performed the second round of inductive coding on the remaining 85% of the documents. Afterward, both the first and second authored deliberated in the codes that emerge during these inductive rounds.

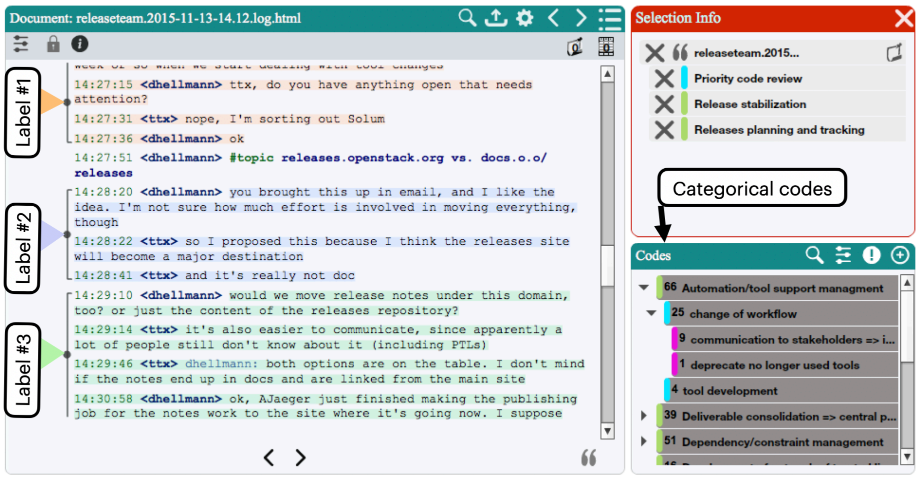
Eventually, a final abstraction of nine high-level codes and their low-level codes emerged, each representing a significant release synchronization activity. The children of each high-level activity correspond to different tasks associated with it. Furthermore, the excerpts related to each code provide illustrations of each activity and responsibility.

Deductive Coding. Furthermore, to triangulate the results obtained with the IRC weekly logs of the two OpenStack releases, we asked an external researcher to independently apply deductive coding on all the weekly-meeting logs of the Ocata release between Mitaka and Queens.

This deductive coding happened in three iterations. The external researcher found some additional 'stand-alone' low-level code in the first iteration. However, after a deliberations session, both the external researcher and the first author agreed that the low-level code fits under the existing high-level code. As the coding continues (the second iteration), the external researcher applies more labels from the codebook to the text deductively. After submitted the blind coding file, both the first author and the external researcher examined and agreed that there was no high-level code. However, we classified some codes under different high-level categories. Thus the calculated IRR gives a score of $\kappa = 0.86$. At the end of the deductive coding, the external researcher improved the labeling and reorganized the code. Both the external researcher and the first author agreed on the new configuration, which gave a perfect IRR score of $\kappa = 1.0$ [31].

Finally, we rearranged the existing low-level codes in the hierarchy, as shown in the Affinity diagram in Figure 6. Based on the activities' hierarchy and the corresponding IRC chat excerpts, we identified dependencies between activities. These are relations according to which a given activity precedes another activity in time (sequential dependency) or makes another activity more natural to perform (support dependency). Our full coding results are available at our replication package online [37].

Fig. 5 Excerpt of a weekly meeting IRC log with labels (tags) applied on three sections/-paragraphs of the text, and code hierarchy shown on the right.



```
14:27:15 <dhellmann> ttx, do you have anything open that needs
attention?
14:27:31 <ttx> nope, I'm sorting out Solum
14:27:36 <dhellmann> ok
14:27:51 <dhellmann> #topic releases.openstack.org vs. docs.o.o/
releases
14:28:20 <dhellmann> you brought this up in email, and I like the
idea. I'm not sure how much effort is involved in moving everything,
though
14:28:22 <ttx> so I proposed this because I think the releases site
will become a major destination
14:28:41 <ttx> and it's really not doc
14:29:10 <dhellmann> would we move release notes under this domain,
too? or just the content of the releases repository?
14:29:14 <ttx> it's also easier to communicate, since apparently a
lot of people still don't know about it (including PTLs)
14:29:46 <ttx> dhellmann: both options are on the table. I don't mind
if the notes end up in docs and are linked from the main site
14:30:58 <dhellmann> ok, AJaeger just finished making the publishing
job for the notes work to the site where it's going now. I suppose
```

releaseteam.2015...
- Priority code review
- Release stabilization
- Releases planning and tracking

- 66 Automation/tool support managment
  - 25 change of workflow
    - 9 communication to stakeholders => i...
    - 1 deprecate no longer used tools
  - 4 tool development
- 39 Deliverable consolidation => central p...
- 51 Dependency/constraint management

2.5 Preliminary Analysis of Catalogue of Activities

Conjecture 1

The required effort by release team coordination relates to the distribution/flow of release synchronization activities, which interact with each other to produce a synchronized release.

Fig. 6 Affinity diagram showing five of the nine high-level abstractions of the IRC weekly chat logs. Complete diagram is available online [37]

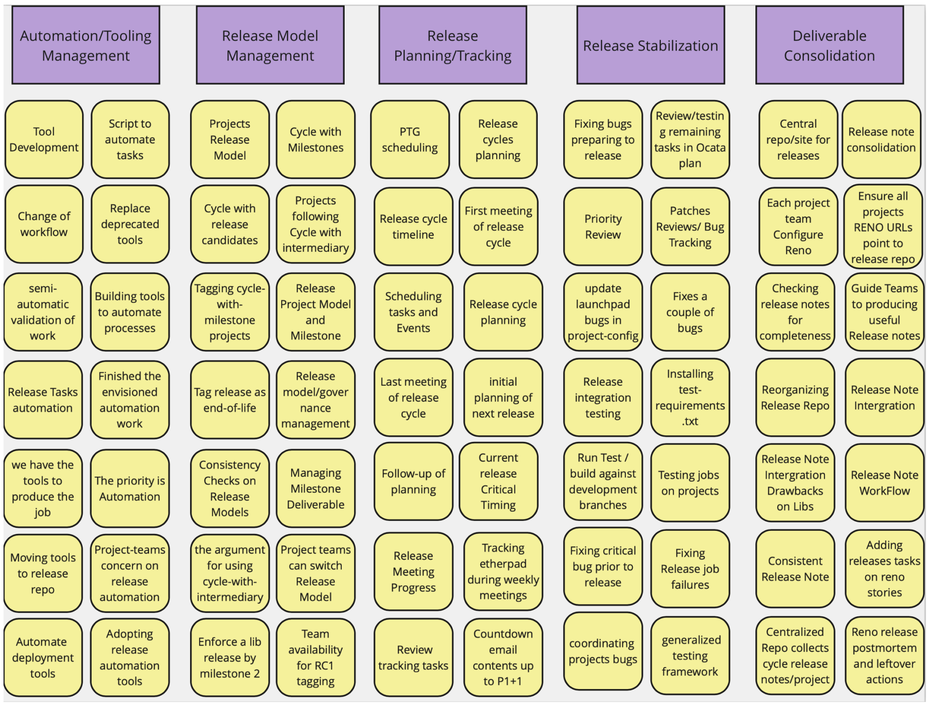
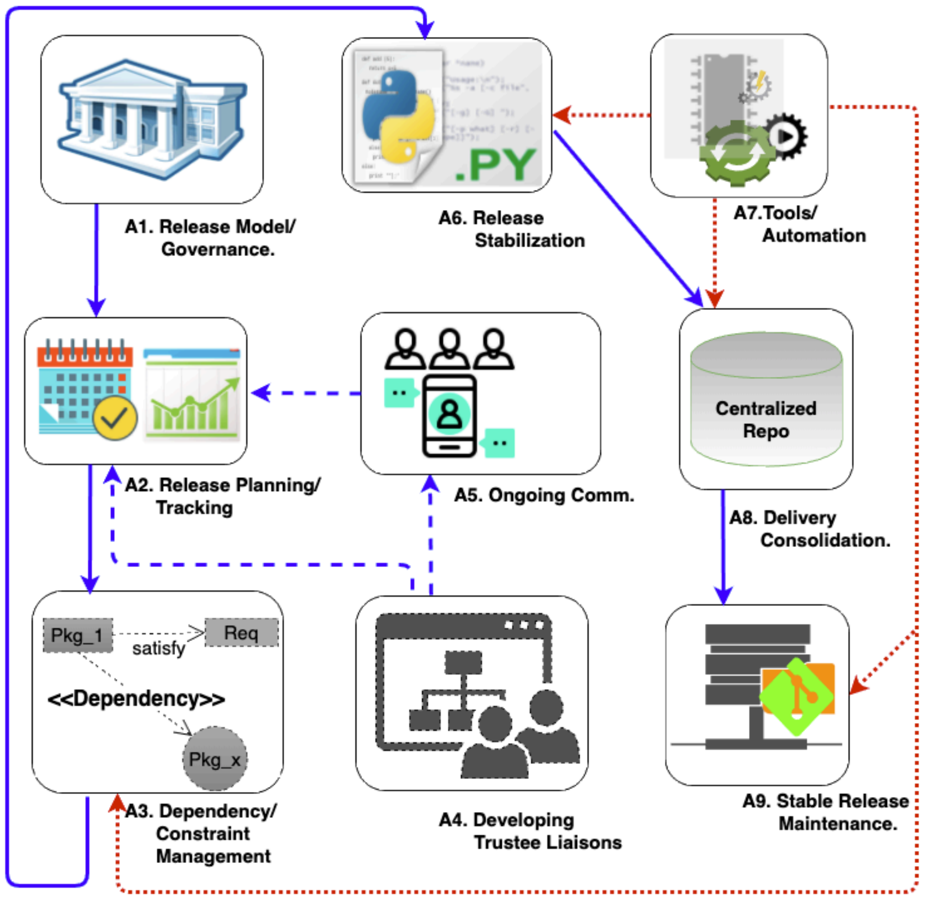

Automation/Tooling Management
Tool Development
Script to automate tasks
Change of workflow
Replace deprecated tools
semi-automatic validation of work
Building tools to automate processes
Release Tasks automation
Finished the envisioned automation work
we have the tools to produce the job
The priority is Automation
Moving tools to release repo
Project-teams concern on release automation
Automate deployment tools
Adopting release automation tools
Release Model Management
Projects Release Model
Cycle with Milestones
Cycle with release candidates
Projects following Cycle with intermediary
Tagging cycle-with-milestone projects
Release Project Model and Milestone
Tag release as end-of-life
Release model/governance management
Consistency Checks on Release Models
Managing Milestone Deliverable
the argument for using cycle-with-intermediary
Project teams can switch Release Model
Enforce a lib release by milestone 2
Team availability for RC1 tagging
Release Planning/Tracking
PTG scheduling
Release cycles planning
Release cycle timeline
First meeting of release cycle
Scheduling tasks and Events
Release cycle planning
Last meeting of release cycle
initial planning of next release
Follow-up of planning
Current release Critical Timing
Release Meeting Progress
Tracking etherpad during weekly meetings
Review tracking tasks
Countdown email contents up to P1+1
Release Stabilization
Fixing bugs preparing to release
Review/testing remaining tasks in Ocata plan
Priority Review
Patches Reviews/ Bug Tracking
update launchpad bugs in project-config
Fixes a couple of bugs
Release integration testing
Installing test-requirements.txt
Run Test / build against development branches
Testing jobs on projects
Fixing critical bug prior to release
Fixing Release job failures
coordinating projects bugs
generalized testing framework
Deliverable Consolidation
Central repo/site for releases
Release note consolidation
Each project team Configure Reno
Ensure all projects RENO URLs point to release repo
Checking release notes for completeness
Guide Teams to producing useful Release notes
Reorganizing Release Repo
Release Note Intergration
Release Note Intergration Drawbacks on Libs
Release Note WorkFlow
Consistent Release Note
Adding releases tasks on reno stories
Centralized Repo collects cycle release notes/project
Reno release postmortem and leftover actions

Table 1 Overview of the release synchronization activities and their distributions across the Mitaka, Ocata, and Queens releases. Data for Cycle Highlights Management (A10) was not available (N/A) in our coding results since A10 is an activity that we identified after the studied Queens release.

ID	Release synchronization activity	$\%_{\text{Mitaka}}$	$\%_{\text{Ocata}}$	$\%_{\text{Queens}}$
A1	Release Model Management	48	46	44
A2	Release Planning and Tracking	100	100	100
A3	Dependency Management	84	70	72
A4	Establishing a Network of Trusted Liaisons	36	34	40
A5	Establishing Communication Channels	48	40	40
A6	Release Stabilization	52	53	56
A7	Tools/Automation Management	80	83	84
A8	Deliverable Consolidation	76	75	80
A9	Stable Release Maintenance	40	49	48
A10	Cycle Highlights Management	N/A	N/A	N/A

Fig. 7 High-level dependency among release activities. Regular lines (Blue) indicate sequential activities, while the dashed lines (Blue) show social activities among A2, A4, and A5, meanwhile, the dashed lines (Red) show activities that are supported by Tools/Automation (A7)


A1. Release Model/ Governance.
A6. Release Stabilization
.PY
A7.Tools/ Automation
A2. Release Planning/ Tracking
A5. Ongoing Comm.
Centralized Repo
A8. Delivery Consolidation.
Pkg_1
Req
satisfy
<<Dependency>>
Pkg_x
A3. Dependency/ Constraint Management
A4. Developing Trustee Liaisons
A9. Stable Release Maintenance.

Observation: Conjecture 1

Figure 7 provides an overview of the identified release synchronization activities, as well as their high-level dependencies. Table 1 compares the distribution of the prevalence of the identified activities between the Mitaka, Ocata, and Queens releases of OpenStack. For example, the Release Model Management (A1) activity was discussed in 48% of the Mitaka, 46% of the Ocata, and 44% of the Queens weekly meetings. Release Planning and Tracking (A2), Dependency Management (A3), Automation Management (A7), and Deliverable Consolidation (A8) are the four most discussed activities across the weekly meetings. Despite some small fluctuations in percentages, the distribution of activities is similar in the three releases. Moreover, the volume of activities such as Release Model Management (A1) and Stable Release Maintenance (A9) is higher either initially or towards the release cycle. For example, A1 happens more at the beginning and A9 towards the end. Only A2 maintains a harmonious flow of activities in the Mitaka, Ocata, and Queens releases. Instead, A3, A5 (Establishing Communication Channels), and A9 seem to have significant differences in prevalence: Queens has less A3, A5, and more A9, Ocata also has less A5, A5, and more A9, while Mitaka has more A3, A5, and less A9. ***Therefore, when the release team applies more effort on A3 and A5, A9 will require less effort to synchronize a given release.***

2.6 Understanding the OpenStack release strategies

To better understand the different release models (strategies) supported by OpenStack, we cloned the release team's Git repository[11]. We explored the various deliverables within the master branch of the release repository "releases/deliverables[12]."

The release repository contains all the releases from Austin to the current development release (Victoria). We parsed the *.yaml* file of all the projects found in each release repository and extracted the 'release-model' type that each project uses for a given release. Based on this information, we show a visualization of all the different release models used within the projects of each OpenStack release in Figure 8. *"Cycle-with-intermediary"* and *"Cycle-with-rc"* are the two most common release models, used by 1,746 (58.97%) and 319 (10.77%) OpenStack project/cross-project teams. We found evidence of three legacy release models (*"Cycle-with-milestones"*, *"Cycle-trailing"* and *"Cycle-automatic"*) which have been replaced by either the *"Cycle-with-intermediary"* or *"Cycle-with-rc"* models in recent releases. The reader should note that we omit three strategies that are not managed by the release team from Figure 8: Untagged (52 projects), Abandoned (19 projects), and Independent (86 projects).

Release Strategies (#Projects managed by Strategy)

1. Cycle-with-intermediary (**1746**)
2. Cycle-with-rc (**319**) } **627**
3. Cycle-with-milestones (**308**) }
4. Cycle-trailing (**371**)
5. Cycle-automatic (**60**)
6. Untagged (**52**) ✗ }
7. Abandoned (**19**) ✗ } Strategies not managed by the release team.
8. Independent (**86**) ✗ }

2.7 Validation using Interviews

During the OpenStack summit/PTG in Berlin, Germany (November 13–18, 2018), the first author conducted an interview study with eight practitioners (four release team members and four project team members) from the OpenStack ecosystem. We selected the participants for the interview based on their longevity in the OpenStack ecosystem. Participants must have actively participated in at least four major OpenStack releases, including this study's three releases (Mitaka, Ocata, and Queens).

We present the demography of the interviewed participants in Table 2, and for confidentiality, we use mnemonics to refer the interviewees. Among the four project team members, three were female, whereas all four release team members were male. The release team has only one female member. However, she didn't meet the criteria for inclusion. We observed that the release team members have similar characteristics. Thus, we build a persona for the release team at OpenStack and explain the skill-sets in more detail in section 4. The average experience for a release team member and project team member is eight years and seven years. Each member gained his or her expertise in more than one OpenStack project team, even though some release team members got experience with other ecosystems before joining OpenStack (e.g., in the Linux kernel, etc.).

[11] `shorturl.at/eiGKY`

[12] `https://opendev.org/openstack/releases/src/branch/master/deliverables`

Fig. 8 Release team manages multiple release strategies per release cycle from Austin, to the current development release cycle; Victoria

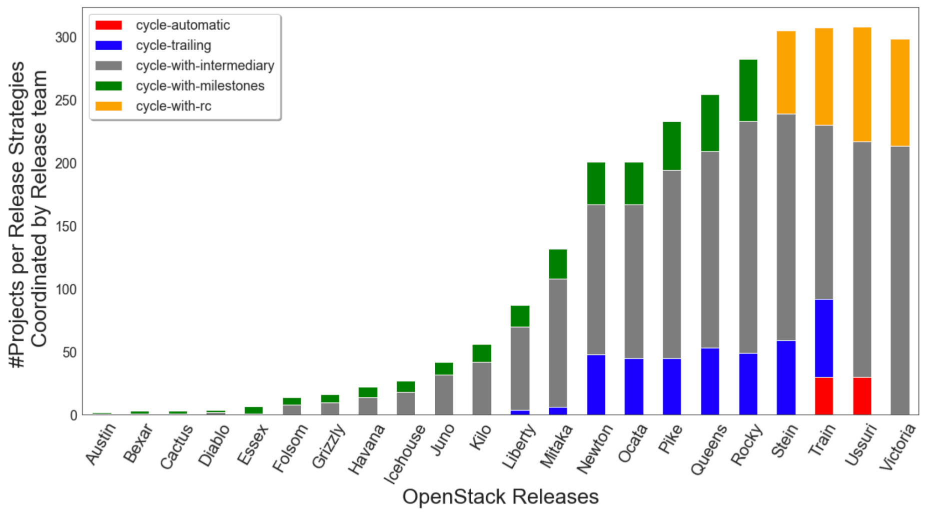


Table 2 Demographics of interviewed OpenStack experts. The experts either belong to the release or project teams and were **active** during the studied period.

ID	region	Gender	Hired	Team	Active	#years
RP1	Americas	♂	Yes	Release	Yes	Six
RP2	Europe	♂	Yes	Release	Yes	Nine
RP3	Asia	♂	Yes	Release	Yes	Nine
RP4	Europe	♂	Yes	Release	Yes	Eight
PP1	Europe	♂	Yes	Swift	Yes	Ten
PP2	Americas	♀	Yes	Ironic	Yes	Seven
PP3	Americas	♀	Yes	Manila	Yes	Six
PP4	Asia	♀	Yes	Nova	Yes	Six

We performed a semi-structured interview with each participant, lasting about 40 minutes. The first author asked the interviewees for activities they deemed essential in the daily work of the release team and then discussed in more detail those activities of Figure 7 that the interviewees did not mention. For each activity, we asked the interviewees for their feedback and experience. In terms of correctness, all interviewees anonymously agreed that the nine identified activities were correct. In terms of completeness, one additional activity, Cycle Highlight Management (A10), which we did not identify during our qualitative analysis, was brought to our attention by the participants. This activity was introduced during the next release after Queens and explained why the IRC logs that we studied did not provide evidence for this activity. In Section 3, we discuss Cycle highlight Management activity (A10.) in detail.

3 Catalogue of Release Synchronization Activities

The catalog of release synchronization activities presented in this section covers the release activities performed by the OpenStack release team, as identified from their weekly meeting logs. To separate fundamental (OpenStack-independent) principles from OpenStack-specific details, we document the activities using the following structured format:
What: Brief outline of the goal of the activity.
Why: Short description of the rationale behind the activity.
How: The major tasks involved in the activity based on the OpenStack observations.
Examples: Illustration (s) of discussions about this activity in the context of OpenStack; a complete list of all examples identified in the log files, as well as all the codes that we tagged, are available online [37].
Expert: Insights and feedback provided by the interviewed OpenStack experts.

A1.Release Model (Strategy) Management

What: Each project that wants its releases integrated into the ecosystem release should select the central release team's desired way to coordinate this integration. Once selected, the project accepts to honor the responsibilities and rights related to the chosen release model, while the central release team pledges to guide the projects according to the selected model.

Why: Given that each project has its release road map and (possibly) dates, some projects might prefer to keep their course, and only at set times (determined by activity $A2$) synchronize with the release team (release-based management model). Other projects might prefer closer follow-up through predetermined deadlines of milestones and final releases (time-based management model). Hence, an ecosystem release team should offer different release models that the project teams can choose. Once chosen, the model acts as a contract for the release interaction between the project and the release team.

Having a complex ecosystem with multiple release models is a non-trivial problem to understand, even among ecosystem community members. Release team members often have to explain to community members why different release models exist, how they differ from each other, and which one to use in what context[13].

How: The choice of a release model is not immutable. If a project wishes to change its development model in a subsequent release cycle, they must inform the release team, which is in charge of tracking and enforcing the model.

> OpenStack— *All official OpenStack projects' deliverables should go through the Release Management team to produce releases*[14].

Ecosystem projects typically choose a release model based on their near future challenges and goals. For example, time-based release models like OpenStack's "cycle-with-rc" (formerly: "cycle-with-milestones") require a project to deliver a specified number of milestones or release candidates (RC) within an *agreed upon schedule* established by the release team at the start of the cycle. The last milestone coincides with the project's feature freeze [38], after which the release team cannot add any new features in the upcoming

[13] `shorturl.at/ckmDH`
[14] `shorturl.at/jrFXZ`

ecosystem release before the stabilization activity (fixing critical bugs before the upcoming release). Figure 9 provides a graphical overview of this release model.

On the other hand, feature-based models such as OpenStack's "*cycle-with-intermediary* releases" (see Figure 10) are more flexible since they allow the projects to make their formal releases as they see fit, without any imposed milestone/release candidate deadlines nor feature freeze. Instead, the projects should ensure that their current internal release in the last month of the ecosystem's release cycle is sufficiently stable to be included as the project's official release. For example, in Figure 10, the blue tags in the current release cycle indicate that 1.2.3 is the final release of the project for the current release. Any newer feature proposed after 1.2.2 will be carried forward to the next release cycle, and, instead, the project is encouraged to focus on making bug fixes for 1.2.2.

Time-based models are typically preferred by projects that aim at doing fewer releases (one-release-per-cycle), especially most core projects that are reaching maturity [39]. Meanwhile, ***feature-based models are popular with projects (modules) that aim at doing several intermediary releases before the final ecosystem release.***

We observed that all OpenStack libraries opt into the feature-based model since it strongly encourages frequent releases, even without rigid rules, and invites a more collaborative interaction with the release team instead of the more policing interaction of the time-based model. OpenStack's current time-based model is already a watered-down version of the previous "cycle-with-milestones" model, which deprecated since September 2018[15], and had two additional milestones (see the greyed-out tags in Figure 9). Overall, the release-based model is the most popular in OpenStack.

Examples: `Missing deadlines.` A recurrent topic discussed by the release team is their frustration with projects that fail to meet the deadlines set out by their selected release model. While the first reaction to deadline misses is to send reminders, these still did not prevent projects from missing their milestones. A suggestion was made to unsubscribe such projects from the forcefully "*cycle-with-milestones*" model. In such situations, the release team has a policy to "force a release" on projects when they miss an important deadline, enabling the team to take the previous stable release of a project and "force" it on the current release for that project.
`Meeting deadlines.` Conversely, on February 9, 2018[16], the release team showed their satisfaction with the progress of projects in resolving stabilization bugs. Some release team members even commented that the Queens release cycle had been one of the most successful in terms of commitment towards deliverable deadlines.

Experts: The interviewed release team members unanimously agreed with our analysis, while the interviewed project members provided additional insights. For example, two people stressed that the current 6-month release cycle is too short, especially for participating companies with typical timelines (delays due to code review and external dependencies are widespread). They have many other objectives apart from OpenStack. An inevitable decline in new feature development was observed, which might also be due to too short a cycle time.

While the missing of deadlines was acknowledged, the intermediate deadlines of the "*cycle-with-milestones*," enabled both the release and project teams tremendously in predicting the outcome of the ecosystem release: *"Yes, for example, that was one of our problems with a patch that I was developing with my team was a feature that was hard to understand.*

[15] `shorturl.at/pvZ59`
[16] releaseteam.2018–02–09–15.00.log.html

For many other team members, and since we missed this deadline . . . , which meant that we already saw that from a technical perspective, it's going to take a couple of weeks or months to implement the given feature. Still, we already knew that it just wouldn't make into it until the next release" (**PP2**).

Meanwhile, concerning why projects follow different release models, another expert explicitly motivates why they follow the "*cycle-with-intermediary*" release model: *". . . So, it's some sort of 'flexible' I will say, but not a strict model. We try to get things done the most that we can get done, and then if we fail to deliver what we need to deliver, we do retrospectives and try to address the needs for the next cycle.*" (**PP3**).

Fig. 9 Six-month release candidate-based model ("cycle-with-rc").

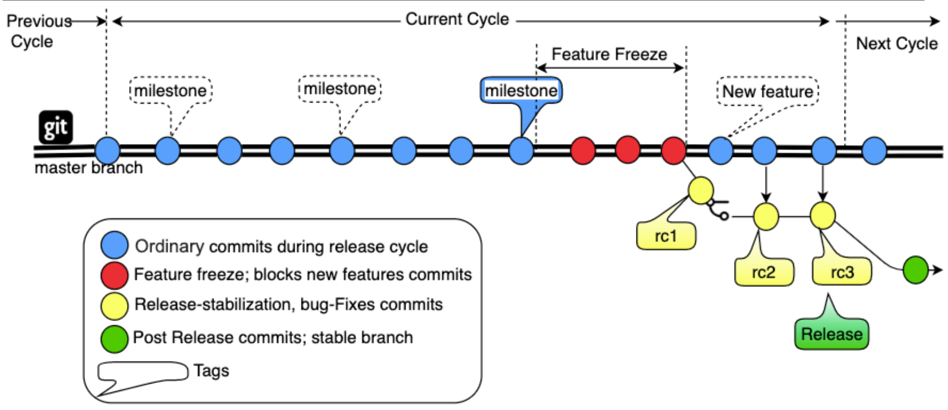
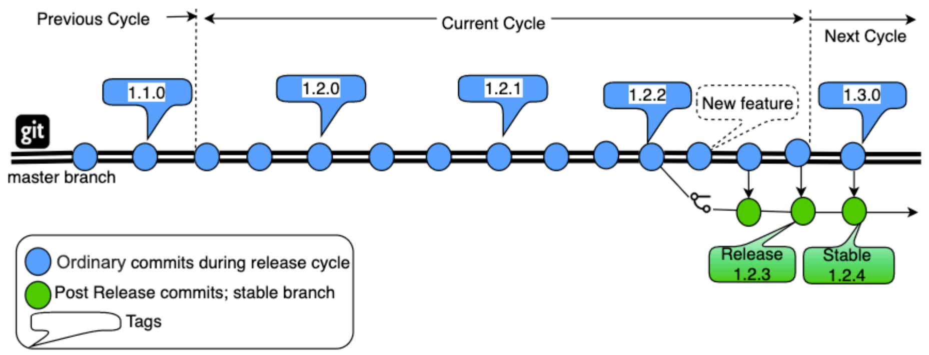

Previous Cycle
Current Cycle
Next Cycle
Feature Freeze
milestone
milestone
milestone
New feature
git
master branch
rc1
rc2
rc3
Release
Ordinary commits during release cycle
Feature freeze; blocks new features commits
Release-stabilization, bug-Fixes commits
Post Release commits; stable branch
Tags

Fig. 10 Six-month intermediary release model ("cycle-with-intermediary").



A2.Release Planning and Tracking

What: Coordination and tracking of the ecosystem's release roadmap.

Why: Since the release model chosen in `Release Model Management` only stipulates the high-level obligations of a project in terms of deadlines and hand-off of releases to the maintenance team, the activity of release planning is necessary to consider the specific requirements of the project's release from the perspective of the ecosystem release. This release plan prioritizes the essential features and bugs requested by the ecosystem's users. It serves as a formal agreement or commitment between projects and the release team in terms of *content* of the project's release and the precise *timing* of a given release (e.g., assigning exact dates to the milestones of a time-based release model). To ensure the plan works out, the release team should track the ecosystem projects' progress over time, possibly adjusting strategies as needed.

How: OpenStack (OIF) organizes several events to bring together a wide variety of community members. For example, the main ecosystem event (Open Infrastructure summit) happens bi-annually and is open for everyone in the ecosystem, including newcomers and vendors. Usually, OpenStack announces the most recent ecosystem release to the general public during this summit.
Some other events (Forum and Project Teams Gathering (PTG), formerly known as "the design summit") are more technical and focus on getting feedback from operators and proposing the requirements and design for future OpenStack releases.

The Forum[17] is a co-located event to the main OpenStack summit. It is organized around DevOps (developers+operators) to discuss the "What" of the OpenStack design on specific features or issues and gather extensive feedback from Devs and Ops. Moreover, Forums focus on strategic discussions and marks the next release cycle's planning phase. For example, Devs can ask specific questions to Ops on their user-experience, and Ops provide feedback from the previous OpenStack release. Also, Ops provide suggestions on cross-community collaboration relating to new features in the upcoming release. Typically, there are three types of discussion sessions: Project-specific sessions, Strategic session on community-wide concerns, and Cross-project sessions.

On the other hand, the PTG[18] happens right after the summit, organized for developers/-contributors from different cross-project and projects teams who are engaged in improving the future releases of OpenStack. PTGs discusses the "HOW" in implementing the forum's suggestions; getting an agreement, building trust among projects/cross-projects teams w.r.t the release teams, assigning work items, and getting work done.

PTG schedules are dynamic and span across three days, permitting project teams to be flexible and productive. During the PTG, the release team starts an online collaborative document (Etherpad[19] in the case of OpenStack), specifying each project's required tasks, the release team manages for the entire next release cycle. These tasks, listed as todos, range from high priority ecosystem features to improvements in release automation or changes to the release process. Liaisons across different projects also integrate their project's proposed project plan. Finally, each task is assigned to a release team member to follow up.

[17] `shorturl.at/glrwx`
[18] `shorturl.at/jmpF2`
[19] `shorturl.at/lnGW5`

Once the new release cycle starts, the release team goes over each active task in the release plan document at the beginning of each weekly meeting. The assigned team member or liaison reports back and suggests follow-up tasks. As project teams implement tasks, the release team mark those tasks as done. Similarly, the release team updates the plan depending on the newly added or modified task. The release plan is a living document that the release team closely monitors during the release cycle.

Finally, on the last day of a release cycle, the release team goes over the plan a final time to ensure that all tasks are completed, with left-over tasks possibly delegated to the next release.

Examples: `Last minute issues.` On April 1, 2016[20], the last but one meeting (R-1) of the Mitaka release was held. A discussion centered on "step-by-step release week actions" in which the release team lead urged some PTLs and Liaisons to have a pre-release meeting and brainstorm on how to process their releases. The release team PTL also noted that some PTLs/Liaisons who were absent from the previous meeting did not do the tasks that they were assigned to do. Slightly, they were raising concerns that were not part of the agenda for the ongoing meeting and were not also documented in the etherpad (tracking document).

Moreover, one other release team member suggested that since there is already a task action in the release etherpad pointing to the last meeting (R-0), everyone should follow the etherpad documentation. As the discussion progresses, the release team PTL said he would need to send out the usual announcement for the last meeting (R-0) earlier than scheduled, and other release team members agreed to do extra hours of work to get things done on time.

`Overwork`. On October 20, 2017[21], a project team liaison wanted the release plan to be modified to meet a pending deadline. Several non-trivial changes should go into the plan that required an integration test to validate whether the changes were correct. However, other members refused because if they modify the release-plan at this point in the release cycle, there would be no guarantee that the rest of the release timeline would be respected. After much back-and-forth, some members stepped up who were willing to work extra hours to fix the problem without delays.

Experts: The interviewees explained that release dates are not determined using strict rules but are established in collaboration with the OpenStack foundation in a pragmatic manner. For example, the release team will "*look for other holidays or major events that are going to be happening and try to pick a good day for the final release [. . .] we didn't want to release things on what was a Friday afternoon. . .* " (**RP1**). Similarly, **RP2** (and **RP4**) stressed that a significant goal of the release plan is to "*make sure everyone knows where we are in the cycle,*" rather than serving just a managerial purpose.

A3.Dependency Management

What: Since, by definition, the individual projects within an ecosystem depend on each other and third-party libraries, the ecosystem's releases need to agree upon and specify these dependencies as well as constraints on eligible versions of each dependency.

Why: Ecosystem projects are inter-dependent by nature [11, 12, 40], which can yield a variety of problems commonly referred to as "dependency hell" [10]. For example, two projects

[20]releaseteam.2016-04-01-14.01.log.html
[21]releaseteam.2017-10-20-15.00.log.html

could depend on different, incompatible versions of a third project. A project could depend on a project that will not be included in the ecosystem release or on a third-party library with an incompatible license or does not reach the desired quality level. Even if project dependencies would be specified, individual projects could churn out new releases faster than foreseen (given that they are following their road map), forcing other projects to keep on updating to newer versions of their API.

How: Ecosystems require their projects to register all their dependencies in a centralized dependency specification (e.g., in OpenStack this specification is stored in a file called "global-requirements.txt"). As is common in open source projects, project version numbering, and hence the dependency specifications[22], use semantic versioning rules [10]. Furthermore, the dependencies in the centralized specification are enforced during ecosystem release testing. Any changes to this file, specifically, addition, removal, or modification of version constraints, need to be reviewed by the release management team.

The latter team also runs tests against unpinned dependencies — the practice of not making explicit the versions of software components depends on, which is not a good practice to encourage. On the other hand, dependencies pinning 'freezes' dependencies, which in turn makes deployment repeatable. If not, different versions of the same software component will run at each re-staged of servers. Also, unpinning dependencies prevent notification of vulnerability. Consequently, this implies that a test on each patch tests two things, (i) changes in that patch, (ii) releases of dependencies. At the project level, each project has its own local sets of dependencies for that project, and this implies that any arbitrary patch is tested on changes in the patch itself and for dependencies.

The management of dependency specifications and review of any changes to them is not necessarily the sole responsibility of the release team. In OpenStack, these two activities are the shared responsibility of the release team and the requirements team, independent of one another. The dependency specification reviews that they perform consider several important questions, such as: "Does the library or project dependency have a (responsive) maintainer?", "Is it still active or deprecated?", "Does it have a compatible license?", "Do the dependency's APIs overlap with or complement existing dependencies?", "Is the library backward-compatible?", etc. In order not to miss dependency changes during the review, tools can automatically flag and track such modifications to the reviewers.

However, what is currently missing in this activity, but seems essential, is any mention of the timing of communication/coordination about dependencies for a given ecosystem. For example, when do projects decide to adopt a new version of another project (which is still under development) and know what version of APIs to use other projects (since even under construction)?

Currently, they do this through project liaisons who updates the cross-project teams weekly on changes to their project that might affect other projects that depend on their deliverables.

Example: `Floating dependency specification` On February 10, 2017[23], the release team tried to resolve the issue of floating dependency specifications. This issue pops up when a maintenance branch is made for a stable ecosystem project release. Often, a project forgets to specify a concrete version number of dependency and instead only refers to the latest version. Since the maintenance branch is dedicated to maintenance activities on the

[22] `shorturl.at/ekBF9`

[23] releaseteam.2017–02–10–15.00.log.html

state of the ecosystem at the time of a specific release, while in the meantime the project continues to evolve, floating dependencies would try to download potentially newer and different dependencies than available at the initial time of release, flagging confusing test errors.

As the discussion progressed, some members proposed alternate dependency requirements. One possible idea was to use local requirements, whereby each project handles a local copy of its dependencies, which are consumed by that project. Yet, this idea was rejected by the release team for fear of incompatibilities. Another idea was to develop a script to validate stable releases (this stable release checked checks for floating dependencies), making sure that a new release of some third-level dependency does not invalidate the constraints change (that goes with every dependency).

This impacts new development: "you can't do this change since it breaks an earlier version under maintenance".

Eventually, a decision was made to have stable project releases refer to the specific version of each of their dependencies in "global-requirements.txt".

> Unpin keyring to unbork twine validates the check. Tim Burke reported that just-released twine 3.0.0 requires keyring$\geqslant$ 15.1, but we have a 3yearold keyring==7.3 pin in releases requirements.
>
> The pin was put to avoid installing a full desktop (through a "DBUS" dependency introduced in >7.3), but this seems to have been fixed between now and then.
>
> Change-Id: I904ea3735daaa8892edc54be4a9611beb66fdfe2
> master
> ttx committed 12 hours ago

Experts: Communication was mentioned to be an essential practice to reduce the impact of changes on dependent projects. The release team members explained how their team could play a vital role in this by warning dependent projects of essential changes ("instructive communication") or organizing online meetings to bring together stakeholders of affected projects ("dialogue").

The project members acknowledged this and also stated *"I feel like we've gotten much better being effective at that kind of communication."*(**PP4**). The release team also explained how their collaboration with the requirements team is crucial to prevent projects from using buggy dependencies.

A4.Establishing a Network of Trusted Liaisons

What: The appointment, assignment, and training of trusted liaisons, specifically, members of individual projects responsible for coordinating their project and the ecosystem release team on all matters related to the inclusion of their project's releases into the ecosystem release.

Why: Given the autonomy of projects inside an ecosystem, the release team needs to interact with representatives of each involved project to coordinate the ecosystem-level releases. This activity covers information exchange, the delegation of technical changes, project members' training about the overall release process and automation, reporting problems, etc.

How: For each project, the ecosystem recommends a single point of contact ("trusted liaison"), which often is the technical lead of a project ("PTL" in OpenStack), but in theory, could be any member suggested by the project or cross-project team. Projects/cross-projects teams announce any changes in the liaison list in time for the upcoming release (e.g., in OpenStack, technical leads are only elected for a given release cycle).

The liaison role is a technically demanding job, with many responsibilities, requiring experienced, skilled people. For example, each liaison must be present in the release team meetings ("#openstack-release" IRC channel in OpenStack) to answer questions about his/her project and be ready to address technical issues. A liaison should also ensure that release-related patches are reviewed on time by their project and should pass on any queries, messages, or training material from/to the release team.

Example: `PTL vs. Liaison.` During several meetings, the performance of current PTL-liaisons is discussed, with opinions ranging from very positive to needing improvement. From these discussions, it is clear that PTLs' implication in the overall release process is one of the criteria for re-election as PTL.

For example, on November 27, 2015[24], one of the PTL-liaisons was eligible to re-run for the upcoming election due to excellent contributions and commitment over the past cycles; you need to have been a perfect liaison to be re-elected. While the incumbent wanted someone else (without experience) to run for election, he finally agreed to serve if elected in office.

On another note, in January 2016[25], the slow progress of work assigned to another liaison was possibly impeding the person's chances to run in the upcoming PTL election. In this case, the liaison ran for election, however, was not elected as the PTL.

Experts: ***RP2*** mentioned that: "*the release liaison [. . .], it's an important role for the release team to ensure that we have a release at the end of the process. And so we have a lot of safeguards in place, and one of them is to make sure that we know who to contact, who can be involved directly, so that's a very integral part [of the release team's responsibilities]*". Each week, the project/cross-project team liaisons suppose to update the release team regarding envisaged changes in the upcoming release.

A5.Establishing Communication Channels

What: Establishment and maintenance of communication channels between projects (liaisons) and release team to ensure effective synchronization of releases.

Why: While the previous activity identified the liaisons that the release team interacts with, dedicated communication channels are needed to propagate all information related to release activities from/to liaisons, for example, to exchange instructions, bug reports, new releases, manage dependencies (A3), etc.

How: The communication channel depends on the kind of information the release team wants to communicate. For example, to communicate instructions, report major issues, etc., liaisons should attend the (online) meetings organized by the release team. Meanwhile, for

[24]releaseteam.2015–11–27–14.34.log.html
[25]releaseteam.2016–01–08–14.00.log.html

more informal contacts, specific chat-rooms, private messages, or mailing lists are used. Project management tools such as Trello (possibly integrated with the issue repository) can enable teams to track work progress.

Apart from conceptual information, ecosystem releases also require the exchange of artifacts with individual projects. Again, ecosystems should encourage the use of different communication channels for different kinds of artifacts. The OpenStack case study showed how loose coupling, supported by automation, is essential. For example, for a project to signal a new release's availability, all it needs to do is use special tags in its Git repository to tag a branch. Release automation then automatically picks up the latest release and automatically starts testing. To exchange project artifacts that need to be consolidated into one ecosystem-level artifact, the release team should follow the process and communication channels of activity A8. For example, to generate the release notes that open source distributions, press, and marketers use.

Examples: `Communication with Stakeholders` On March 25, 2016[26], during one of the weekly release team meetings, there was an ongoing discussion concerning what communication medium the release team should use to reach all different stakeholders, including PTLs since the release team wanted to change their meeting days from Mondays to Fridays. Thus, the release team needed confirmation from all the stakeholders before they could make the change.

Some members suggested that the release team lead send an email to the mailing list addresses and ping (to start a conversation with) the relevant people on the IRC channel. As the discussion progresses, we observe that the release team used all the communication media to send the community's message.

Furthermore, one release team member expressed frustration because he could not reach a particular technical committee (TC) member assigned to work with the release team member on an upcoming project team gathering (PTG) meeting. Two weeks later, the release team member could ping the TC member over IRC and invite her to follow the IRC release channel for onward discussion concerning the PTG meeting.

Experts: As mentioned earlier for A3, all interviewees uniformly agreed on the importance of communication and using the right channels. For example, regarding the use of Git tags for communicating (the contents of) a new release, **RP1** added that "*that process helps us do things like making sure that teams are communicating correctly, . . . and then chasing up the people that don't . . . that's the main: communication, communication, communication*". On the other hand, IRC was preferred for in-person communication with and amongst the release team.

A6.Release Stabilization

What: Coordination of testing efforts, and fixing of integration and other urgent issues on the upcoming ecosystem release.

Why: Once individual projects have handed off their release to the release team, e.g., after feature freeze in a "cycle-with-milestones" model (see A1), the release team is in charge of testing the upcoming release. Release stabilization is a significant activity since its input is

[26]releaseteam.2016–03–25–14.00.log.html

the set of releases of the individual projects planned for inclusion in the forthcoming ecosystem release. At the same time, its output should be a well-polished product, ready for the end-user.

How: The majority of this activity's work involves integration and automatic system testing and bug fixing of identified issues, typically under tight time pressure (due to the impending deadline). For example, these tests would find conflicts between project dependencies or incompatible API changes (cf. A3).

Projects are tested first at the project level; then, the release team does the integration testing and debugging. In general, three types of testing happen here to stabilize the release: unit, functional, and integration testing. In case of problems, the release team collaborates with the projects whose tested release has an issue.

Example: `Priority bugs.` On December 18th, 2015[27], release team members discussed the "priority list" of open bugs blocking the final release. Several tasks were assigned to members, most of which were related to testing the Reno consolidation machinery (see A8). Some projects had difficulties integrating Reno properly. As a result, these braked some functionalities in those projects, and the release team had to run several test cases, including integration tests on all those projects. Finally, the problems were all resolved, and the final release was successful.

Experts: ***RP4*** stated that "*the stabilization and automation activities are the most time-consuming activities for us*". In particular, the release team spent most of its time reviewing and **testing** submitted features of projects (such features were also discussed earlier on in IRC, after that sent to ZUUL (CI) for automated testing). Due to the amount of effort involved, the release team encouraged projects to attract and train community members to become skilled at helping out with project-level testing.

A7. Automation Management

What: Determining the need for, the coordination of, and the management of ecosystem-wide tools for release automation, such as CI, system, and performance testing and deployment/release.

Why: Release engineering is known to be a highly automated domain [16, 17], since many activities, such as the compilation, testing, deployment, etc. are repetitive and would be error-prone when left to manual interaction. While this holds for the release process of individual projects, it also applies to the ecosystem's release process as a whole, since a central release engineering team would be unable to manually release and coordinate the individual ecosystem projects taking into account the dependency constraints between them. Without which, this would be a tedious, error-prone job for the central release team. Delegating such automation to the individual projects is not scalable either, given that they lack an ecosystem-wide view.

How: During the weekly release team meetings, automation-related issues are initially raised and discussed at the project-level. Those discussions typically try to flesh out the rough re-

[27] releaseteam.2015–12–18–14.21.log.html

quirements for new automation. If an existing open-source tool is available to satisfy these requirements, the issue can be resolved locally (by the release team). If not, the project-level discussion sometimes brainstorms potential designs for new automation before escalating the issue to the ecosystem-level Infrastructure (Infra) team responsible for building the infrastructure around the ecosystem. However, the release team builds the automation that runs release jobs.

Indeed, the ecosystem-wide infrastructure has integrated some of the tools that the release team conceived and developed locally to facilitate the release automation process. Further, the release team contributed some of these tools to the Python community. In contrast, the release team host other tools on their repository[28], which is publicly available. The release team then brings up such issues in one of their regular meetings to make an executive decision, such as discarding the plans, starting work on a new tool, or evaluating the need for a particular tool.

Overall, it's preferable to use standard tools and platforms such as Ansible, Docker, or Kubernetes over custom scripts and tools. Standard tools enable better compatibility with the full range of technology used by individual projects and reduce development effort. Only when standard tools do not work out are custom ones explored. For example, OpenStack initially used Jenkins as its CI platform, but this did not scale to the massive volume of commits merged per day and the enormous scale of tests aimed at running on Jenkins. Therefore, the ecosystem made a conscious decision to implement a new CI platform, Zuul. While Zuul was custom-built for OpenStack, the ecosystem recently made it open source to become a standard CI environment.

When the ecosystem decides to pursue a new tool, the release team members consult with the infra team responsible for developing the ecosystem infrastructures. At the same time, the impacted liaisons keep track of progress. In case a tool exists that could do the job, the liaisons instead perform a search online and analyze the identified tool's acceptability. Once the resulting tool (either developed or acquired) has been evaluated and shown to be functional, it's time to discuss its integration into the regular release team meeting's ecosystem release process. Since the developed tools' actual operation is not the release team's responsibility, the infrastructure team, members of the latter, are also involved in those meetings.

Examples: `Standard vs. Custom technology.` On June 5th, 2015[29], the release team members were exploring ways to improve the performance of several tools. Tools developed in-house are mixtures of scripting languages and perform slower [41]. Furthermore, there was an alternative plan to find more scalable, standard libraries to replace those scripting-tools. The release team also discussed a tool for tracking milestones. However, some members suggested that the tool was not idempotent; specifically, repeated executions each time would yield different results. Consequently, the PTL requested a release team member to migrate the Bash tool to Python to reuse existing libraries.

`Temporal measures vs. Full automation` On January 8th, 2016[30], the release team noticed that they were stuck with the automation process to release milestone-2 for the Mitaka release of a particular project (Barbican) since the project was breaking the Zuul automation pipeline on integration test. While the release team and the Infra teams did the

[28]`shorturl.at/cdgn7`
[29]releaseteam.2015–06–05–13.01.log.html
[30]releaseteam.2016–01–08–14.00.log.html

best to resolve the problem, certain activities like release note consolidation and milestone announcements would have to be done manually for the time being.
Experts: **RP3** summed things up nicely by stating that "*. . . the main [responsibility of the release team] is communication, communication, communication, and then the second provides the tools and the automation behind the process.*"

A8.Deliverable Consolidation

What: Consolidating artifacts produced by individual projects into a centralized, ecosystem-level repository managed by the release team.

Why: While the autonomy of individual projects to set out and pursue their release schedule is one of the strengths of an ecosystem, end-users should not be assumed to keep track of each project's release cycle, versioning numbering, dependencies, etc. Instead, they expect to install and use a vetted combination of compatible project versions. The same principle applies to more mundane artifacts, such as having one source of documentation for the released ecosystem, one set of release notes, or even one central download site.

How: The key to consolidating ecosystem artifacts is to provide a central repository for the individual projects to submit their finished deliverables and guidance and tool support to integrate this submission into their workflow. The latter is the most significant challenge since each project uses its local process and automation, hence loose coupling with the central repository is preferred. Typically, either an API is provided to the projects or a standard location where projects can put a deliverable for the release team's tool-chain to detect and integrate it automatically.

While this consolidation applies to a wide range of deliverables, these do not necessarily require custom tools or processes to release notes and documentation from actual project releases. For example, the OpenStack release team designed a generic process for consolidating deliverables. During the release team weekly meetings, this process comes up often, which only relied on a central Git repository with a configuration YAML file listing for each sub-project, the current version (commit hash) whose deliverables are part of the ecosystem release.

Example:`Release Notes.` On Nov. 11th, 2017[31], the Reno infrastructure for consolidation of release notes was discussed. Reno logically groups release notes of individual projects, which are arranged into categories/sub-categories that automatically triage to the right reno category[32], for example, new features, bug fixes, etc.

This grouping occurs immediately after project teams add the notes into their project repository, then Reno pushes and integrates them into a central Git repository. Reno manages release notes for OpenStack deliverables in the same git repository where the project's source code is in, the central repo with all projects' code. Therefore, Reno can track the history of files across all branches and all revisions of the ecosystem. For more implementation details, we refer to [22].

While the Reno tool was ready to go into production at the meeting time, most of the discussion involved informing and training individual projects about Reno. Also, whether

[31]releaseteam.2017–11–17–15.00.log.html
[32]`shorturl.at/czNRS`

to perform the adoption of Reno in parallel with the training or afterward. The release team finally decided that the training and roll-out of Reno should happen concurrently. Based on the analysis of the log files, this happens to be a common practice at OpenStack. Whenever there is a new tool (in-house or third-party), the release team usually organizes training parallel to project members to use the tool.

Experts: **RP1** mentioned that "*we don't want the release team to manually mark version numbers on a bunch of release notes or anything like that instead, and we want to handle all that stuff automatically*" and that "*Reno has increased the 'number' of release notes.*" To validate the latter, we analyzed the online Reno repository. OpenStack launched Reno in 2015,[33] and in July 2018, over 13,000 release notes were gathered by Reno for over 300 projects, which is far beyond the experts' initial expectation and more than all the release notes written before Reno.

A9.Stable Release Maintenance

What: Coordinating maintenance activities after an ecosystem release, resulting in bug fix (minor) releases.

Why: While work is underway on a new release, the previous release should be kept stable for as long as initially planned. Thus, bugs that slipped through the release stabilization activity or post-release bugs (reported by end-users) have high priority. Special attention goes to reported vulnerabilities, which need urgent resolution.

How: Reported bugs go to the concerned projects where actual fixing happens. Meanwhile, the release team is in the perfect position to coordinate the resolution process. Moreover, the release team applies stricter rules about stable releases of projects on their stable branches and should have a review step.

Project teams focused on issues related to their projects because the projects know the internals of their code best and because they typically will resolve such bugs first in the code of their next release (to ensure that they do not have regressions), then back-port the relevant fixes to the stable branch (es) of the previous release(s). The release team delegates reported bugs to the project liaisons, who propagate them within their respective project/cross-project teams.

In OpenStack, stable branches typically are maintained for about 18 months, but this can go beyond (extended maintenance) in case of strong demand. A release enters end-of-life (EOL), specifically, requires no further maintenance activities. Maintenance has three phases:

1. first-six-months: all bug fixes accepted;
2. six to 12 months: only critical bug fixes and security vulnerability patches accepted;
3. 12 to 18 months: extended maintenance, accepting only security patches.

Example: `Whose responsibility.` On May 6, 2016[34], some projects asked the release team to manage stable release management for them. The request was partly due to confusion about the extent to which maintenance of stable releases was covered by the release

[33] `shorturl.at/AFVY9`

[34] releaseteam.2016–05–06–14.02.log.html

model that project teams selected and partly due to those projects not being managed by the release team at that time. The majority of the release team members refused the request, preferring to delegate this responsibility to a "stable team" comprising each project's liaisons (as recommended by the OpenStack governance policy).

Experts: **RP1** clarified that "*we enforce stricter rules about stable releases of projects on their stable branches and having a review step [. . .] allows us to make sure that the stable maintenance team has an opportunity to look at the release and approve the release in addition to the release team.*"

As mentioned in A1, the scope of the release model goes beyond the actual release. The exact duration that the release team provides maintenance is undefined: "we will keep two or more existing releases in maintained State and then an undefined number in extended maintenance and then a very few numbers and un-maintained, and then the rest would be end-of-life" (**PP1**).

A10. Cycle highlights Management

Prologue: The release team experts unanimously agreed about the nine identified synchronization activities. They observed that shortly after this paper's studied period, the release team adopted a new activity, Cycle Highlights Management (A10.), which targets non-technical users dealing with the release process, specifically, public relations, marketing, media, etc. Instead of ignoring this activity (since it came towards the end of the Queens release cycle), we decided to write up still the tenth activity profile, based on the experts' feedback and other background information we could find. In other words, we could not derive this activity from the studied IRC logs and release plans.

What: The release team aims at providing a high-level summary of the next release's features to stakeholders who are neither developers nor operators, such as sales/product managers, media/press, marketers, and users. Thus, this summary should provide a concise overview of the upcoming new features and other changes in a non-technical manner.

Why: Today's evolution towards more rapid releases and continuous delivery [16, 42] bears the risk of alienating users since they might become overloaded with ecosystem releases whose contributions are unclear and overwhelming. It could lead users to not update to the newer ecosystem releases, which would increase the ecosystem organization's support costs and the risk for vulnerabilities/bugs.

In particular, in the OpenStack ecosystem, a higher volume of emails from non-technical users was received on the community mailing list concerning what to expect in upcoming releases. For some time, each project team responded independently to these community requests, leading to inconsistencies or sometimes no response from project teams. Hence, the ecosystem decided to provide an ecosystem-wide high-level overview of changes, called "cycle highlights."

How: Starting towards the end of the Queens release (see Figure 3), specifically, right after the period studied in this paper, the OpenStack release team introduced a template that the individual projects should use to present cycle highlights Management to non-technical users. The template is stored under *deliverables/$RELEASE/$PROJECT.yaml* in the *openstack/release* repository, and contains the following fields:

1. A summary of fewer than two lines in length.
2. What are the changes/new features, etc.?
3. What benefits do these changes bring to the user?

At the beginning of each release cycle, specifically during the three-day PTG/design submit, the release team requires all project team leads or liaisons to fill out the template and submit their resulting highlights.

The release team encourages project teams to do this before the rc1 deadline so that it will be able to edit and help projects write consistent highlights that showcase their work.

OpenStack introduced A.10 towards the end of the Queens release cycle. Moreover, Train was the most recent release at the time we were writing this paper. Besides, we had already studied the IRC logs, and those logs didn't contain any information regarding A.10. Further, information concerning A.10 didn't appear on the PTG sessions before the Queens release. It didn't feature anywhere else in this paper because it was the most recent information when we were writing this paper.

Example: `Whose responsibility?`
The first author of this paper attended a 3-days **Train** in-person PTG event to understand better the design submits. During one of the sessions, there was an ongoing discussion concerning the cycle highlight Management of some project teams that didn't send their cycle highlights to the release team in time.

"*Cycle-highlights collection needs to start earlier … otherwise marketing asks for info too early and we can't provide them the info without collecting it manually.*[35]"(Anonymous1)

In response to this suggestion from the foundation members, the release team lead requested the dedicated person-in-charge of A10 to adjust the window frame for A10 and make sure the project teams' cut-off date to submit their highlights comes before the feature freeze period (FF). The foundation would be aware by the **rc1** deadline of any projects that have not yet sent their highlight.

Later, based on the discussion that the release team documented on the etherpad during the Train PTG[36], we learned that about 4% of the project teams did not submit their highlight during the Train release cycle, and we verified this observation by counting the projects that didn't submit their cycle-highlights (it is missing) from the Train release page[37]

Experts: We asked the release team experts to explain why cycle highlight Management is an important release team activity during the interview session: "… *The release team gives cycle highlights to the public relation and marketing staff within the foundation. They use that to talk to analysts. They prepare a presentation at each summit that includes that kind of information sometimes, and it makes it into the keynotes stage at the summit. It's not helping people who are not directly involved in the development understand the significant work that happened for each release.*"(RP1)

Fig. 11 Ecosystem release with projects following different release strategies in a release cycle.

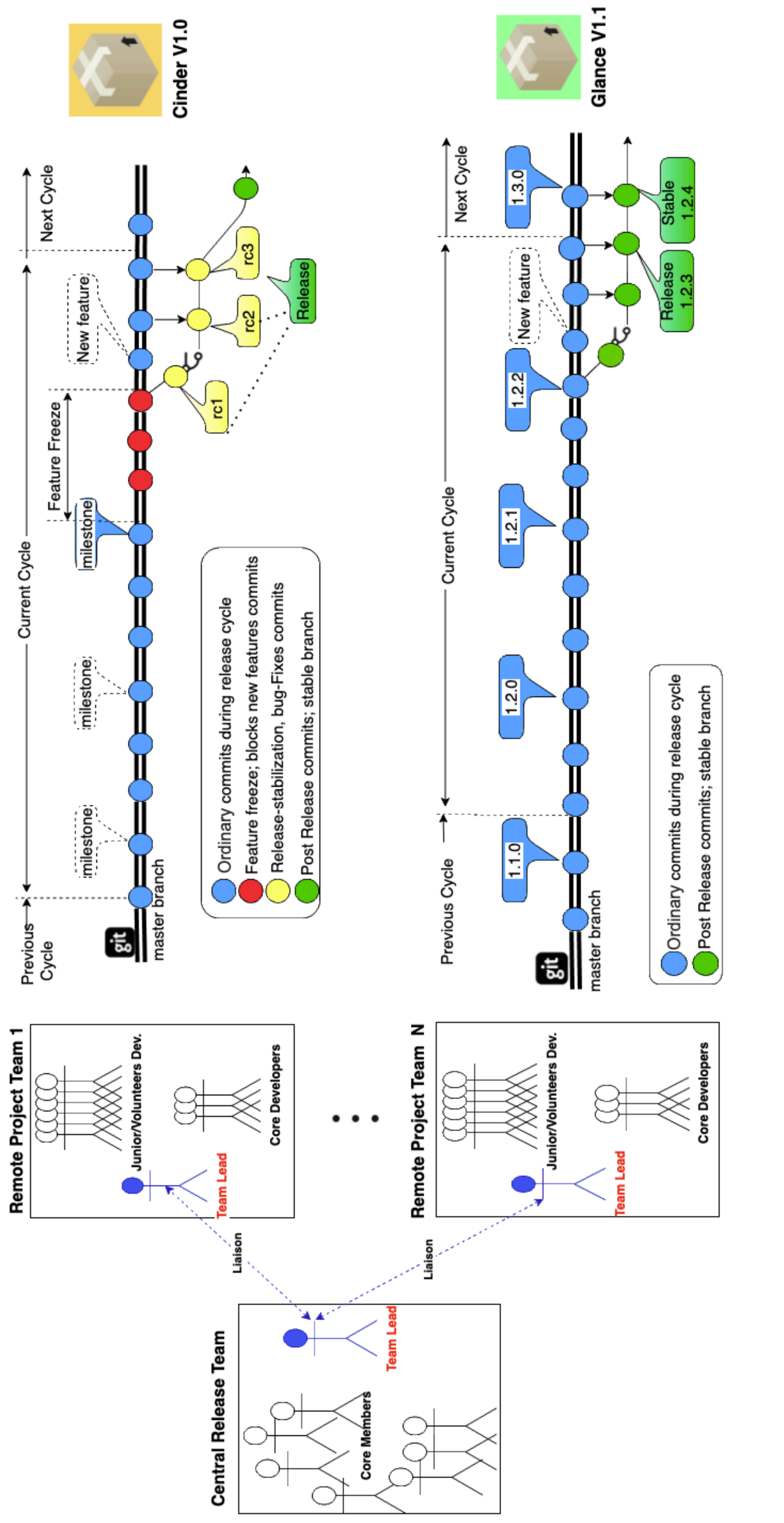


Fig. 12 Excerpt from a "*.yaml*" configuration file, showing the Swift project on Rocky release following the cycle-with-intermediary release models.

```
---
launchpad: swift
release-model: cycle-with-intermediary
team: swift
type: service
repository-settings:
  openstack/swift: {}
branches:
  - name: feature/deep-review
    location:
      openstack/swift: 331339246228167dfab61690327fc0c3f2fab8af
  - name: stable/rocky
    location: 2.19.0
releases:
  - version: 2.18.0
    projects:
      - repo: openstack/swift
        hash: f270466de363499894317b7c671f65e8a912bd53
  - version: 2.19.0
    projects:
      - repo: openstack/swift
        hash: f0472f1f7975957fc31cb8c123aa82ee47848645
  - version: 2.19.1
    projects:
      - repo: openstack/swift
        hash: 3d2d954107d676e48acb81069639eed15ead5713
  - version: 2.19.2
    projects:
      - repo: openstack/swift
        hash: 5aa4c5d88fec98cf4ca0536c511c61759e865ec9
  - version: rocky-em
    projects:
      - repo: openstack/swift
        hash: 5aa4c5d88fec98cf4ca0536c511c61759e865ec9
cycle-highlights:
  - Added an S3 API compatibility layer, so clients can use S3 clients
    to talk to a Swift cluster.
  - Added container sharding, an operator controlled feature that may
    be used to shard very large container databases into a number of
    smaller shard containers. This mitigates the issues with one large
    DB by distributing the data across multiple smaller databases throughout
    the cluster.
  - TempURLs now support IP range restrictions.
  - The trivial keymaster and the KMIP keymaster now support multiple
    root encryption secrets to enable key rotation.
  - Improved performance of many consistency daemon processes.
  - Added support for the HTTP PROXY protocol to allow for accurate
    client IP address logging when the connection is routed through
    external systems.
release-notes: https://docs.openstack.org/releasenotes/swift/rocky.html
```

4 Discussion and Implications

4.1 Discussion of our findings

Co-existence of multiple OpenStack release models

Indeed, in every release cycle, the release team has to manage multiple release models simultaneously, which is at the heart of the complexity of ecosystem release synchronization,

[35] `shorturl.at/hsFP1`
[36] `shorturl.at/ctDG2`
[37] `shorturl.at/xGOQ1`

as shown in Figure 11. For example, Cinder follows the Cycle-with-rc model; meanwhile, Glance follows the Cycle-with-intermediary model. The central release team has to keep track of both projects throughout the development cycle.

Over 2961 projects/modules exist in the OpenStack release team repository across all releases. However, 105 (3.5%) are not coordinated by the release team; they do not follow the 6-months release cycle, even though, from time-to-time, they produce releases independently without the coordination of the release team. The majority of the project teams (96.44%) are coordinated by the release team and follow one of five ecosystem models.

- **Model-1**: Cycle-with-intermediary is the most prominent release model and over time 1,746 (58.97%) of OpenStack project/cross-project teams have been using this model.
- **Model-2**: Cycle-with-rc, which replaces the Cycle-with-milestones model after the Rocky release and ranks the second most popular model within the project/cross-project teams. In OpenStack's history, over 319 (10.77%) project teams have signed in to use this model.

  OpenStack Legacy release models:
  - **Model-3**: Cycle-with-milestones, now replaced with cycle-with-rc, from Austin to Rocky releases, model-3 grows in popularity and then stop existing after Rocky release. Actually, it merges to cycle-with-rc from Stein and onward releases. In total over 308 (10.42%) of OpenStack project/cross-project have used this model.
  - **Model-4**: Cycle-trailing was introduced in the Liberty release and continues till the Train release; later, it was replaced by the cycle-with-intermediary release models. Throughout its existence, 371 (12.53%) of OpenStack projects/cross-project teams used model-4.
  - **Model-5**: Cycle-automatic, this release model was also replaced by the cycle-with-intermediary model, and over 60 (2%) of project teams have used this model-5 in only two major releases; Train and Ussuri.

- **Model-6**: Untagged

  (*The release team uses some continuous integration (CI) and automation tools only from source and never tag releases, but need to create stable branches. For example, the Grenade projects*[38] *from the Quality Assurance team uses the 'Untagged' as release-model. Hence, we found 52 instances (1.75%) where different CI tools (projects) across all OpenStack releases are marked 'Untagged'.*)

- **Model-7**: Abandoned

  (*We observe that with time, some OpenStack projects/modules became obsolete and produce no new future releases. These projects can either be 'abandoned' or merged with other project/module. In the history of OpenStack, 19 (0.64%) projects/modules are marked 'Abandon'*)

- **Model-8**: Independent

  Usually, projects that are independently released can either be in two possible conditions such as 'Abandon' or switch release model to a release-team managed model. In total 105 (3.5%) of projects have been released independently.

[38] `shorturl.at/gGJY8`

Moreover, our study of the release team communication (IRC meeting logs) and release plans (Etherpad docs) shows that the ten release synchronization activities apply to all the release models. Besides, most OpenStack ecosystem projects consider the first two models: Model-1 (Cycle-with-intermediary) and Model-2 (Cycle-with-rc). Therefore, no matter the release model that a project uses, we can study their release synchronization activities.

In particular, Model-1 (Feature-based model) is typical with projects that are doing several 'intermediary' releases before the final ecosystem release, and this model favors projects with many new features coming in.

Meanwhile, Model-2 (Time-based model) is typically for projects experiencing a low volume of new features coming in and aim at doing one major release per-development-cycle with one or more release candidates (RC). Notably, most core projects that are reaching maturity prefer this model.

Perception of OpenStack Release Process

> RP2:"... The release team's primary role is to synchronize the ecosystem releases in collaboration with the project teams that are released managed, ...
>
> The second role of the release team is to provide the **tooling and automation** to support the release process."

Consequently, it comes as no surprise that *A7 (Automation/tooling)* is pivotal to the release process in complex ecosystems. We found evidence suggesting that the release team has automated more than 80% of the release engineering activities. These tools are open source and hosted online in OpenStack's repositories[39], while some of these tools t1[40], t2[41], and t3[42]. have been contributed to the Python community.

On the other hand, some of the project practitioners we interviewed, who also happen to be project team leads (PTL) in their respective projects, affirm that from December 2015 on, some OpenStack projects started experiencing a decrease in new features. Therefore, since fewer new features have been coming in, these projects suggested switching from a six-month cycle time to a nine- or even twelve-month cycle time.

> PP2:"... there's not a ton of new features coming in anymore; it's a lot of like stabilization and optimization. I think that nine months would be fine because there are not as many new things coming in anymore that we need to like release to get out the door for collaborators and stuff."

In contrast, some other projects were still experiencing a high volume of new features coming in (and they requested a shorter, three-month cycle time), the release team settled on the current six-month release cycle. Hence, the six-month cycle time is not a dogma or magical number but rather a balance to satisfy the different ecosystem projects' needs and concerns.

[39] `shorturl.at/iAGMN`
[40] `shorturl.at/elxW9`
[41] `shorturl.at/bistz`
[42] `shorturl.at/bxOZ9`

Dependencies between Activities

We observed two types of dependencies, precisely, sequential and social, among the nine activities listed in Figure 7.

First, **sequential activities** exist among Release Models Management (A1) $\Longrightarrow$ Release Planning and Tracking (A2) $\Longrightarrow$ Dependency Management (A3) $\Longrightarrow$ Release Stabilization (A6) $\Longrightarrow$ Deliverable Consolidation (A8) $\Longrightarrow$ Stable Release Maintenance (A9), in this order respectively. The sequential dependencies follow the chronology of the release process, which is supported by Automation/tools Management (A7) except for A1 and A2, as RP2 stated above. A1 is the only activity that occurs only once, at the beginning of each release cycle. Also, no other activity relates to A1, sequentially or socially, unlike the different activities.

Second, the **social activities** Establishing a Network of Trusted Liaisons (A4) and Establishing Communication Channels (A5) ensure that the release team is in constant communication with the project teams at every stage of the release process. Thus, it facilitates ongoing communication with the project teams by ensuring that they have a contact person with the release team, helping the release team keep track of the ongoing progress during a given cycle.

Equally important, Cycle highlights Management (A10.) targets the non-technical audience of what has changed since the last releases or the new changes expected in the current release. OpenStack introduced this practice towards the end of the Queens's release. We found evidence of A10 in all the OpenStack projects located in the release team repository. For example, Figure 12 shows A10 at the bottom of the configuration *.ymal* file for the Swift project with six new features that the Swift project added in the Rocky release. The release team then collects Cycle highlights Management (A10.) from all cross-projects/projects/modules and submits them to the marketing team. Coordinating A10 activity requires skills in writing documentation in a precise and concise manner.

4.2 Implication of our findings

4.2.1 Implications to Ecosystem

Despite advances in synchronizing ecosystem releases and how release strategies co-exist in complex ecosystems, release processes still fail and become irksome to end-users. Retrospectively, several ecosystems could have mitigated some of the challenges they suffered if our research's findings were available.

A single activity not well performed has a critical impact on the entire release process. As previously discussed, there are sequential and social dependencies between the observed release synchronization activities. Thus, the absence or poor execution of a single activity can derail the release team's efforts and various projects.

For example, in February 2012, Eclipse release manager sent a message[43] to the cross-project teams

[43] `shorturl.at/etH78`

He declared with the most profound regret the failure to release the first release candidate (RC1) of that series. While the release manager had been sending 'notes,' they were not regular and timely. As the discussion progresses, other release-team members retrospectively pointed out that they too were running behind schedule. For example, a release engineer responsible for running specific scripts could not deliver the results before scheduled. Therefore, this shows a lack of proper planning and tracking (A1) of release activities. We observe a critical impact of this poor planning in the latter stages of the release process as dependency issues (A3) start to manifest — the release team members were dealing with dependency issues, with over 80 libraries that were missing. They described this as "worst problems" because some features were disabled due to the dependency issues. These need to be resolved urgently to prevent future deliverables from failing in the next RC2.

In another example in the Gnome ecosystem, we observe that effective communication with the network of trusted liaisons (A4 and A5) is very crucial. In 2011, many prominent open source developers (including Linus Torvalds[44]) discontinued[45] from using GNOME[46] because of core features that the release team removed without formally informing the project teams liaisons on any of the communication channels[47], leading to irritating end-users' experience[48]. Indeed, we noticed that GNOME did not make proper use of some of the synchronized activities. The GNOME release team could have created a dedicated channel with the liaisons from project teams. Maybe this could have helped the GNOME release team circulate information about the 'removed feature.' This observation is also captured in a statement[49] by one member of the release team member:

> "However, the "keep things simple and stable" meme got taken too far. Proposals to add functionality got shot down. Modules couldn't be integrated. In trying to keep things simple and stable and polish existing things rather than creating new ones, a loose and small group of people were inadvertently alienating those who wanted to do new kinds of development in Gnome. There was a lot of discomforts because Gnome seemed stagnant . . . " — Federico Mena Quintero, April 2012.

The findings from this research could have helped Eclipse and GNOME improve their release process, supposed they had our results earlier. *They could learn from OpenStack how to plan and track the release process throughout the developmental cycle, how to manage their dependencies, and how to establish effective communication channels with project liaisons.*

These examples demonstrate what can go wrong in even mature ecosystems. Therefore, mature and young ecosystems need to know about our release synchronization activities to make informed decisions and avoid/reduce such issues. Our findings can help ecosystems practitioners to coordinate cross-project teams to synchronize releases properly.

[44] `shorturl.at/cekG7`
[45] `shorturl.at/nvNQY`
[46] `shorturl.at/rwIJ9`
[47] `shorturl.at/ntP09`
[48] `shorturl.at/yOS19`
[49] `shorturl.at/dpGZ3`

Ecosystems need to continuously re-plan and re-adjust their release strategies. Based on the mishaps in the Gnome ecosystem (detailed above), the GNOME community members have since been continually expressing concern for the improvement of GNOME's release process. This is first observed on GNOME's Wiki page[50], which explicitly states that GNOME is looking for ways to improve its release models from other ecosystems. More recently, prominent community members have been proposing[51] ways to improve the GNOME release process.

A similar evolution is seen in the Eclipse ecosystem. In 2012, Eclipse introduced the "Simultaneous Release"[52] (SimRel) policy as an approach to synchronize releases within the ecosystem. It took six years of continuous discussions and debates before the simultaneous release model was finally implemented in 2018[53]. The main discussion topic in all the threads centered on planing the Eclipse release strategy, which took several rounds of planning and adjustment of the planning.

Despite these advances in improving the release strategy, some Eclipse projects are still not convinced about SimRel. For example, some projects want to follow their schedules; they still need to respect deliverables and deadlines. In contrast, other projects that have reached maturity wonder if they still have to follow SimRel even though they do not have any new release features. Therefore, *our finding will help Eclipse understand that while Planning and Tracking is an essential activity in release synchronization, as some of these activities are sequential*. For example, Eclipse can adopt A1 (Release model management) and implement multiple strategies to take care of different project needs.

Release model management is one of the major challenges that the central release team faces. Even large ecosystems such as Eclipse continuously have to re-plan and re-adjust their release strategies. Therefore, young and even mature ecosystems can use our findings as a starting point or guideline in the evolution of their release strategy."

4.2.2 Implications to Practitioners and Academics

Our findings have several implications for practitioners, especially on managing the release process of a complex ecosystem.

Multiple strategies co-existing. Practitioners and academics could learn from our findings on why and how multiple release strategies co-exist in complex software ecosystems. They could understand why different project/cross-project teams follow their release strategies with a common goal to synchronize their releases at the end of each release cycle, as shown in Figure 11.

Fast cadence ecosystems. Modern ecosystems tend to release with a faster cadence to provide new project features or fixes that meet end-user needs. In a recent talk by Dr.

[50] `shorturl.at/kqrzW`
[51] `shorturl.at/ceAQ8`
[52] `shorturl.at/eEM28`
[53] `shorturl.at/wEKL2`

Yvonne Dittrich[54] on continuous evolution methods and tools in software engineering, a curious participant asked an important question to which there was no clear answer at the time: '*What happens to a fast Cadence ecosystem; pieces of technology that are complex and hard to release quickly? — it's much of a challenge.*"

Our empirical findings provide insights for practitioners and academics. For example, academics now have empirical evidence of why ecosystems that release fast and often may opt for the cycle with intermediary releases; meanwhile, different projects follow different release models dependent on their goals. Therefore, to answer the question asked to Dr. Dittrich, we recommend our findings as a guideline to help the ecosystem towards a solution.

> "Release Engineering *as a Force Multiplier" –John O'Duinn, Director of Release Engineering at Mozilla during his keynote speech at ICSE 2013.*

Release team Persona. During the interview session with release team members, we observed that the release team members (release engineers [43]) collectively play an important role in successfully coordinating projects/cross-project teams throughout release cycles. Furthermore, we observed that memberships to the release team are socio-technically demanding within the ecosystem. The release team members are selected based on their experience with project/cross-project teams, the nature of their contributions across multiple projects (core-contributors), their excellent communication skills, and their willingness to serve the community.

In particular, the first OpenStack release team lead, who was also the lone member of the release team during the early days of OpenStack, declares that he worked with the Linux kernel for many years before joining OpenStack during its creation in 2010. The number of release team members subsequently grew to two before the Mitaka release and eight members during the Queens release cycle. For example, in 2015, the second person who served as the release team lead declares that:

> "*... For the Mitaka cycle, our major goal is to increase the work of automating the release process, so we no longer need to run release scripts by hand. These changes to our release process affect all OpenStack projects, Thierry Carrez and I have been working to publicize them in the community. We spoke in Tokyo, and the video and slides are available online.*" — Doug Hellmann, 2015.

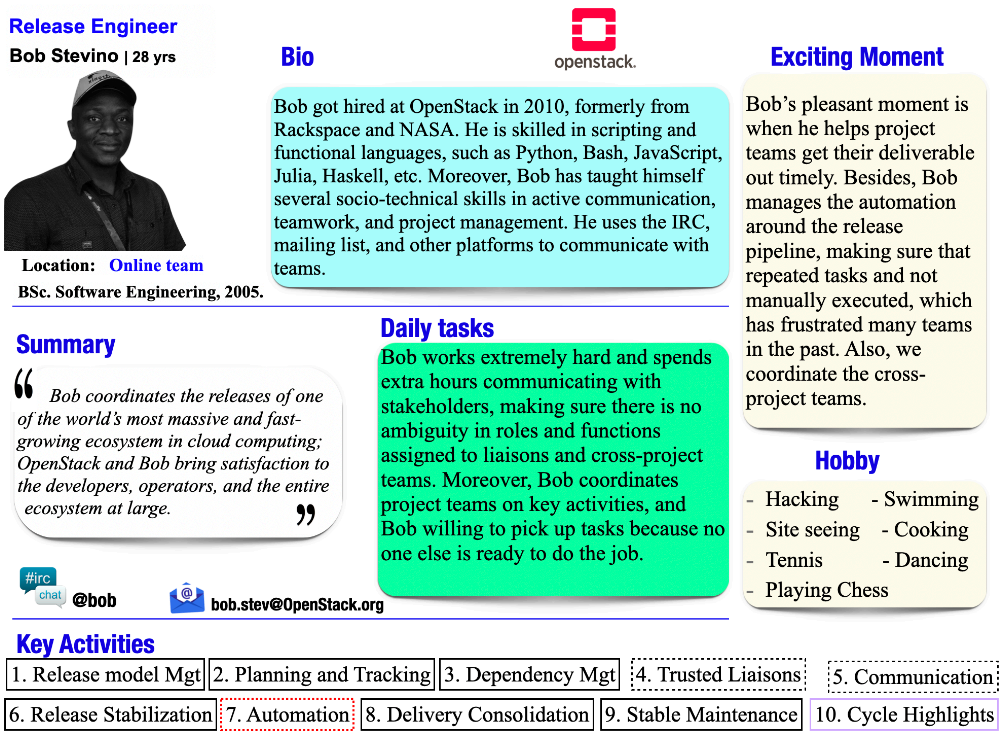
Similarly, other release team members reported that they had served different communities as release engineers or core-developers before coming to OpenStack. Also, we found evidence in the IRC weekly logs that release team members are willing to transfer their knowledge to project team members. For example, when OpenStack introduced Reno, the release team members agreed to train project team members on how Reno works and how to use it in generating release notes.

[54] A 2016 talk at the Microsoft research facility on "Sustaining Software Engineering Ecosystems"(`shorturl.at/moMNP`)

We found evidence that release team members have successfully built tools to facilitate OpenStack releases and have contributed some of those tools to the python community. Also, release team members have been training cross-project teams to use in-house developed tools to facilitate the release synchronization process. Based on their profound skill sets and experiences, which the release team members possess, we have built a persona [44, 45, 46] to describe a typical release team in a complex ecosystem (see Figure 13). Our findings show that the release team is pivotal to an ecosystem release synchronization. Consequently, ecosystem managers and practitioners can use the release team persona to hire/select/train the right skills into the release team.

Persona is a humanized view of who a user (in this context, a release team engineer), such that we can derive valuable outcomes. In particular, creating a 'user' (release engineer) personas starts with research to understand the user (release engineer) — by observing, surveying, or interviewing the users. For details about building a persona in Software Engineering, we refer the reader to the following works [44, 45].

Fig. 13 Release team Personas depicting common values; characteristics and skill-set of an ecosystem Release team.



Release Engineer

Bob Stevino | 28 yrs

Location: Online team

BSc. Software Engineering, 2005.

Bio

openstack.

Bob got hired at OpenStack in 2010, formerly from Rackspace and NASA. He is skilled in scripting and functional languages, such as Python, Bash, JavaScript, Julia, Haskell, etc. Moreover, Bob has taught himself several socio-technical skills in active communication, teamwork, and project management. He uses the IRC, mailing list, and other platforms to communicate with teams.

Exciting Moment

Bob's pleasant moment is when he helps project teams get their deliverable out timely. Besides, Bob manages the automation around the release pipeline, making sure that repeated tasks and not manually executed, which has frustrated many teams in the past. Also, we coordinate the cross-project teams.

Summary

“Bob coordinates the releases of one of the world's most massive and fast-growing ecosystem in cloud computing; OpenStack and Bob bring satisfaction to the developers, operators, and the entire ecosystem at large.”

Daily tasks

Bob works extremely hard and spends extra hours communicating with stakeholders, making sure there is no ambiguity in roles and functions assigned to liaisons and cross-project teams. Moreover, Bob coordinates project teams on key activities, and Bob willing to pick up tasks because no one else is ready to do the job.

Hobby

- Hacking - Swimming
- Site seeing - Cooking
- Tennis - Dancing
- Playing Chess

#irc chat **@bob** **bob.stev@OpenStack.org**

Key Activities

1. Release model Mgt	2. Planning and Tracking	3. Dependency Mgt	4. Trusted Liaisons	5. Communication
6. Release Stabilization	7. Automation	8. Delivery Consolidation	9. Stable Maintenance	10. Cycle Highlights

4.3 Generalization of our findings

4.3.1 Secondary Data Extraction

To generalize our findings, we extracted data from selected ecosystems that support ecosystem-wide releases such as GNOME and Eclipse. These selected communities have projects/cross-projects teams that follow one or more of the observed release strategies supported by OpenStack and archive of its release activities. Using their online documentations, we identify each ecosystem's structure, release strategy, release cycle, and communication medium used by the release team to coordinate releases.

Some of the online documentation that we consulted include: Eclipse[55] [56] [57] [58] and GNOME[59] [60].

Next, We downloaded the archived mailing lists of these ecosystems using custom scripts [37] and Perceval [47]. Then, we ran regular expressions on the archived dataset to search for keywords or patterns related to the release synchronization activities that we found earlier and reported in Table 1. In each high-level activity, we search for the sub-themes (low-level) words, as shown in the affinity diagram in Figure 6. This ensures that we cover a wide range of possibilities of each activity's presence in the extracted text from each ecosystem's mailing list. Moreover, we notice that each ecosystem has its particular technical jargon to express certain concepts; this understanding guided our search of (low-level) technical words related to the release activities. When we are not sure about a particular word (jargon), we search the online documentation, and in rare cases, we email or contact release team members on either the release team mailing list or IRC channel. Afterward, we downloaded the archived mailing list coordinated by the release team for both GNOME and Eclipse ecosystem and perform our search on those archives.

For example, consider a search for the **A7. Automation and Tooling management** activity within the GNOME ecosystem using keywords such as 'script[s][ing],' 'tool[s][ing],' and autom* (for automate, automation, and automating). These keywords return search results that are related to the high-level activity. However, the results are noisy (having false positives). So, we tried different combinations of search queries until we have a meaningful result. Therefore, we randomly select the search result and examine the content. In some cases, repeated patterns of similar search words have nothing to do with discussions on high-level activities. In those cases, we filtered out those patterns and re-ran the search.

Last, we aggregate the number of messages within all threads by date into a *.csv* file. Then we apply data cleaning and pre-processing [48] on the exported *.csv* files to analyze our results (our scripts and data are provided in our replication package [37]). The results of this search could still contain some noise, which could be a threat to our result. We report this in the threat to validity section (Section 6).

To mitigate this noisy result, we could not manually validate all the output results. Therefore, we statistically sampled the search result using a confidence level of 95 % and a confidence interval of 15 % on all the different release synchronization activities in both Gnome and Eclipse. In particular, we manually classified and validated a median of 43 files per

[55] `shorturl.at/kBHX9`
[56] `shorturl.at/jFGQZ`
[57] `shorturl.at/jrvBL`
[58] `shorturl.at/giBKN`
[59] `shorturl.at/oGJX0`
[60] `shorturl.at/adzIJ`

Table 3 Results of manual classifications to validate the regex matching in both Gnome and Eclipse release synchronization activities (A1, A2, ..., A10.), we report the range of precision proportion (PP ± 15 %) per activity on a confidence level of 95% and confidence interval of 15 %.

Activities	**Gnome** PP ± 15 %		**Eclipse** PP ± 15 %	
A1	0.78	1.00	0.76	1.00
A2	0.85	1.00	0.85	1.00
A3	0.85	1.00	0.76	1.00
A4	0.75	1.00	0.83	1.00
A5	0.83	1.00	0.83	1.00
A6	0.83	1.00	0.80	1.00
A7	0.80	1.00	0.78	1.00
A8	0.83	1.00	0.78	1.00
A9	0.78	1.00	0.73	1.00
A10	0.83	1.00	0.80	1.00

release synchronization activity for both GNOME and Eclipse. We used equation 1 to compute the precision proportion (PP) of our sampled date. The sample size (S) comes from a population of each activity. Given S, we manually validate the presence of any of the release synchronization activities. Moreover, if a given text doesn't contain any of our studied synchronization activity, we classified it as false positive (FP) or True positive otherwise. Hence, we compute PP using equation 1, then normalized the result between 0.00 - 1.00 and report the ranges of PP (PP ±15 %) in table 3:

$$\boxed{PP = \frac{S - FP}{S} \pm 15\%} \tag{1}$$

We might have missed an activity that other communities have, which we did not find at OpenStack. Our focus here is to extract only those activities we know about (A1, ..., A10). We report this threat in the threat to the validity section. Moreover, for future work, we plan to examine the release synchronization activities that exist in other open source communities using different communication mediums such as mailing list, IRC, Matrix, Slack, etc.

4.3.2 Findings

In this section, we discuss the generalization of our findings to other ecosystems, in this case, GNOME and Eclipse, and we show the results of a Time Series analysis (TS) [49] on the extracted activities in Figures 14 and 15. Time series is a technique that gleans a trend over some time (monthly in our case). Thus, the y-axis in both figures is the number of times the activity occurs, and the x-axis is the time (in months).

The purpose of this generalization is to corroborate if the release synchronization activities (A1, ..., A10) identified from the OpenStack release team communication channels are also present in other ecosystems. Then, we randomly select some mailing list discussions and follow the threads to understand how a given activity was discussed in both GNOME and Eclipse.

Fig. 14 Time series analysis showing the trends and cyclical w.r.t equation 2 of release synchronization activities across GNOME ecosystem releases

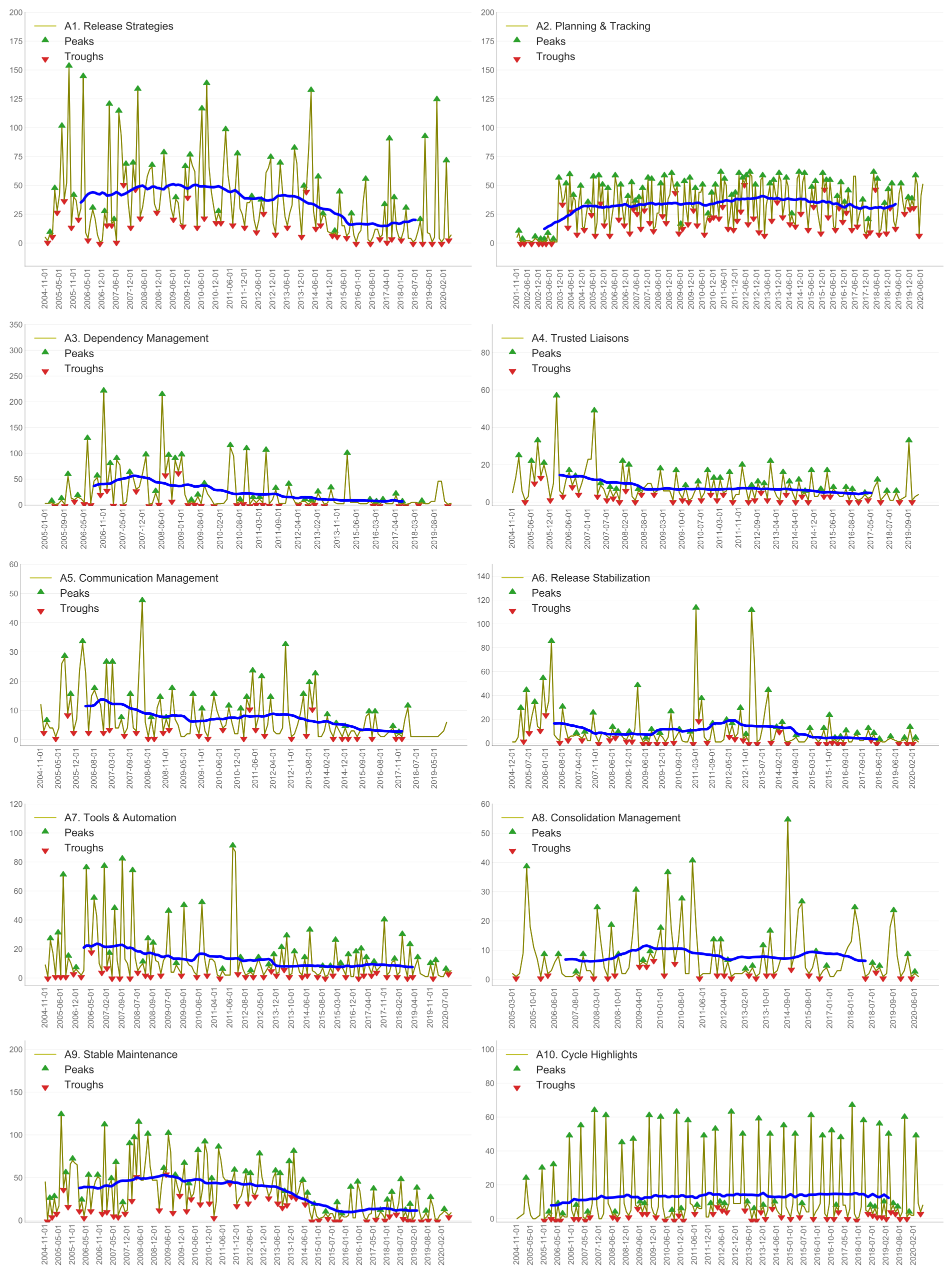
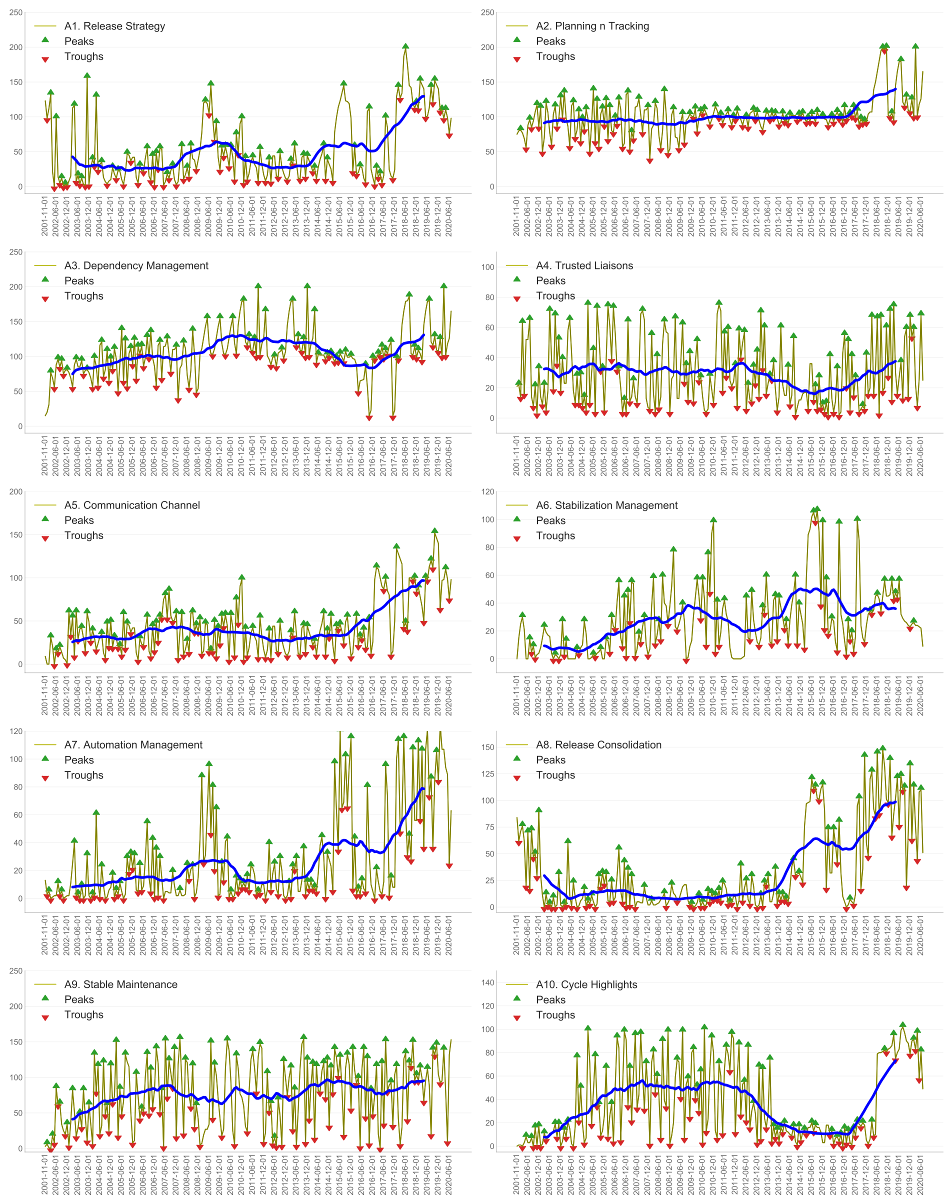


Fig. 15 Time series analysis showing the trends and cyclical w.r.t equation 2 of release synchronization activities across Eclipse ecosystem releases



We use TS with the “Additive Model” to analyze the data. Using the “Statamodels” library in Python. This enabled us to observe and interpret[61] the trend and other TS components (see Figures 14, and 15) $Y_{(t)}$:

Trend ($T_{(t)}$): shown in blue line gives the rolling means at different time scales in our TS data.
Cyclical ($C_{(t)}$): the average value in the series that corresponds to a distinct pattern and shows periodical (but not seasonal) peaks and troughs around the trend, revealing a succession of expansion and contraction phases.
Seasonality ($S_{(t)}$): pattern occurring at specific regular or irregular intervals (it has stable variation concerning timing, direction, and magnitude).
Residuals ($R_{(t)}$) or noise, shows the difference between the observations and the corresponding fitted values. This term introduces randomness in TS model:

Equation 2 shows the mathematical function of the additive TS model.

$$\boxed{\boxed{Y_{(t)}} = \boxed{T_{(t)}+C_{(t)}+S_{(t)}} + \boxed{R_{(t)}}} \tag{2}$$

The $Y_{(t)}$, is the output data, meanwhile $T_{(t)}+C_{(t)}+S_{(t)}$ is the predictable term and $R_{(t)}$ is the random term. All these components are important in observing TS data (trend, cyclical, seasonality, and Residual). The additive model computes the TS with the residual, as shown in Equation 2. The residual gives the difference between the predicted and the actual data. In particular, the residual provides a better indication of our TS model. For example, in our study, the residual was mostly due to random noise, as shown in Figures 16 and 17 (bottom-most graph). Most of the variations are within 10% variation. Our model performance was good and fitted our data well.

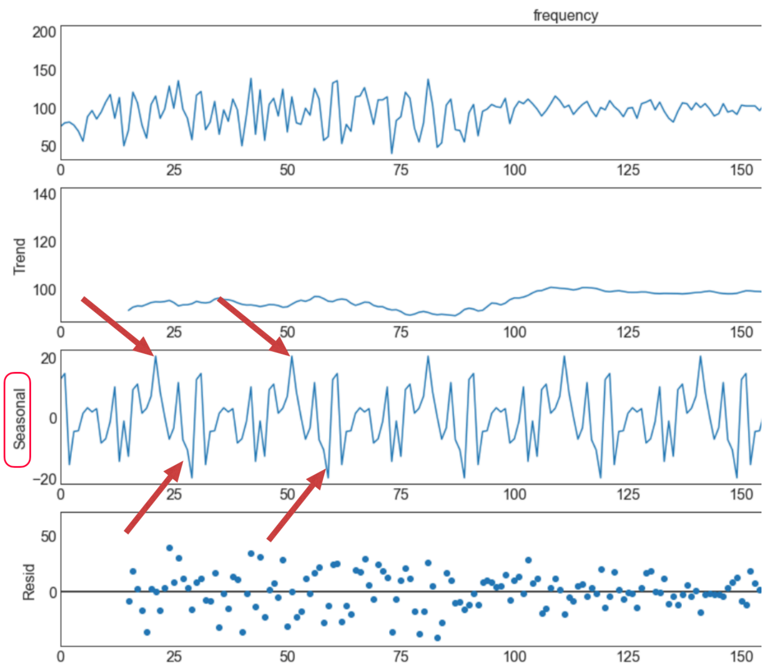
During the additive model training, we noticed that the variability of the peaks and troughs in our data (release synchronized activities A1, . . . , A10) were seasonal; thus, peaks and troughs happened at predictable times. For example, in Figures 17 and Figure 16, we see that the ‘seasonal’ has a repeated pattern on peaks and troughs over time. The trends vary differently depending on a given synchronized activity (A1, . . . , A10). We refer the reader to [50] for more details on time series analysis.

Comparing how activities are mentioned/discussed in both ecosystems. Since all ten activities have seasonality over time, we can say that the release synchronization activities were mentioned or discussed in both ecosystems. For GNOME (Figure 14), we see that although A2 (Release Planning and tracking) was rarely discussed between 2001 and 2003, it gains an almost increasing trend onward. A10 (Cycle highlights Management) was slowly introduced in 2004 and stays virtually constant; both activities were mentioned less initially and maintained a constant flow over time. On the other hand, the trends of the other GNOME’s activities decreases toward the end. A3 (Dependency Management) and A4 (Network of Trusted Liaison) had their maximum variation between 2005-2007 and started decreasing steadily. Besides, A1 (Release Model Management) and A9 (Stable Release Management) had a significant decreasing trend from 2013 onward.

Meanwhile, for the Eclipse ecosystem (see Figure 15), the activities had more of a fluctuating trend. However, we observe a remarkable turning point in release activities from

[61] readers can consult our replication package how the different TS components were model and plotted

Fig. 16 Time series analysis showing seasonality ($S_{(t)}$, repeated patterns; top arrows shows peaks and bottom arrows show troughs), but also ($C_{(t)}$), $T_{(t)}$ and $R_{(t)}$ in the additive training model for A2 in Eclipse ecosystem

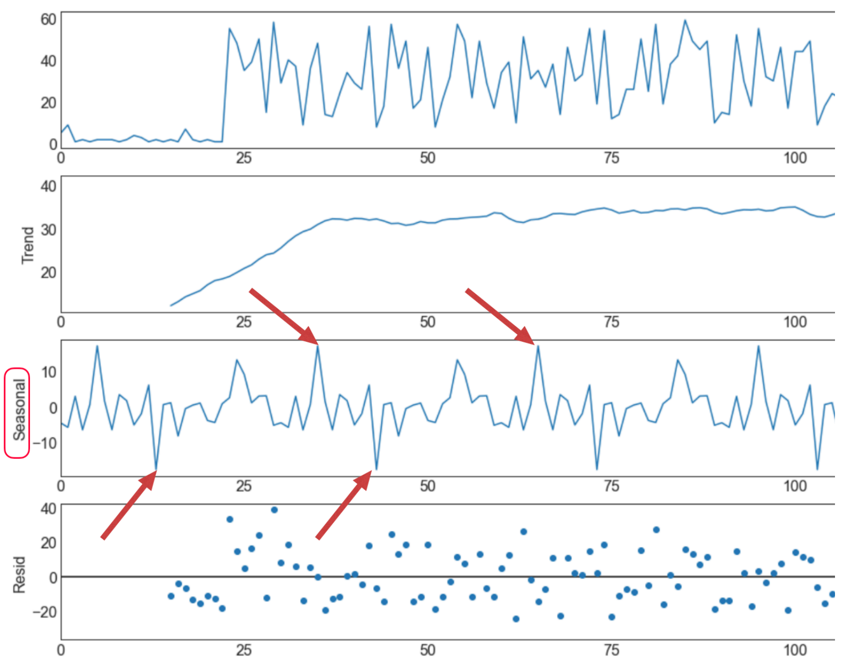


2016-2017; most activities slowed down in late 2016 and started rising again in 2017, except for **A2** (Release Planning and Tracking) that maintains a constant flow over time.

We found evidence that both GNOME and Eclipse have ecosystem release and a central release team that coordinate their respective releases. Moreover, we also found evidence of the studied release synchronization activities (A1, ..., A10) at OpenStack been discussed or mentioned in the context of GNOME and Eclipse. Interestingly, in both ecosystems, these activities have seasonality at particular intervals, which suggest that the activities are discussed following the release process of each ecosystem.

In both cases (GNOME and Eclipse), these trends suggest that release synchronization activities are well present and discussed in both ecosystems in different proportions (variation), similar to OpenStack's case. Therefore, we recommend future works to investigate how the release synchronization activities are coordinated in other ecosystems such as Eclipse and GNOME.

Fig. 17 Time series analysis showing seasonality ($S_{(t)}$, repeated patterns; top arrows shows peaks and bottom arrows show troughs), but also ($C_{(t)}$), $T_{(t)}$ and $R_{(t)}$ in the additive training model for A2 activity in GNOME ecosystem



5 Related Work

5.1 Release Engineering

Adams et al. [9] studied the various activities involved in integrating packages within the Debian, Ubuntu, and FreeBSD open-source software distributions. Like our study, the authors performed a qualitative analysis to identify major activities of the studied process (integration). Yet, they focused on the package release notes instead of the actual communication logs between maintainers. While Adams et al. identified and documented seven major integration activities, typically performed by the maintainers of an individual the open-source project, we instead focus on the synchronization of releases *between projects and the overall ecosystem*.

Adams and McIntosh [16] discuss how modern software projects are released using continuous delivery strategies and the major research challenges in this area. However, they do not consider ecosystem-level releases. The authors suggest that empirical studies are vital to understand and (possibly) enhance release engineering practices, which our paper aims to do for ecosystem releases. Khomh et al. [18] aims at evaluating the relation between faster releases and software quality, which they found a link but at the project level. Nothing is yet known at the ecosystem level. However, our study has not yet evaluated a relationship between the nine release activities and software quality.

Other studies such as Franch and Ruhe [51] studied Release Planning and Tracking and the relation to software quality. Also, Rahman et al. [52] analyzed the relationship between the release stabilization activity and the quality of software, albeit at the level of individual projects. However, our study has not yet evaluated a link between the ten release activities and software quality. We plan to conduct this study in future work.

Poo-Caamano et al. [53] studied the release communication and coordination process of the GNOME ecosystem, which follows a time-based release strategy. Through an exploration of the different communication channels used by developers and the release coordination team to manage GNOME releases, we found four significant challenges facing the GNOME release team: (1) difficulties in coordinating project teams (2) problems in handling the build process, (3) challenges in tracking unplanned changes, and (4) testing the GNOME deliverables. To overcome these challenges, the authors suggest that release team members should have excellent socio-technical communication skills and diverse knowledge of ecosystem-wide concerns. However, these four challenges translate to three synchronized activities in our work (Planning and tracking, Dependency management, and Stabilization) [51], [52], and [40]. Poo-Caamano et al. interviewed 10 GNOME developers to triangulate their findings. Similarly, we also interview eight experts from the OpenStack ecosystem to validate our findings. Nevertheless, the authors did not generalize their work but advocate that future research considers comparing their results against other ecosystems through analytical generalization. Our work fulfills this request by comparing our ten activities against different ecosystems, including GNOME.

Teixeira et al. [22] synthesized online web resources to document the release process and infrastructure used in the famous OpenStack ecosystem. They explore a spectrum of release strategies for OpenStack's projects, from utterly the independent to the product-wide concern, which is fully managed by the central OpenStack release team. Moreover, they emphasize the time-based model at the expense of the other four models that the release team manages. While the release team is deemed essential to achieve 6-month releases of the OpenStack ecosystem product, Teixeira et al. did not focus on the specific synchronization activities and challenges experienced by the OpenStack release team. Mostly because the online web resources that they consulted do not provide information on release synchronization activities, they also extensively explore how OpenStack uses the release notes manager (Reno) across different project teams to automate release notes.

However, this online documentation is limited in capturing the real socio-technical challenges during each release cycle. Teixeira et al. have documented the "what" about the OpenStack ecosystem's rapid release process. They did not study the "how" about the release process. For example, they found evidence online of the different release strategies that co-exist in a complex ecosystem but did not determine why and how they exist in a complex ecosystem such as OpenStack.

Therefore, our study complements theirs by providing empirical evidence on why and how multiple release synchronization strategies co-exist in a complex ecosystem. We also found ten release activities that the release team performs each cycle to synchronize ecosystem releases.

5.2 Software Ecosystems

Zhang et al. [54] studied large companies' involvement in open source ecosystems. The authors investigated the OpenStack ecosystem and the consortium of companies that collaborate to build a sustainable ecosystem. In particular, the authors identify networks of collaboration ("clusters") and four models that collaborative companies within OpenStack have adopted in engaging with the ecosystem. Consequently, they suggest that there are three ways that companies can participate within the ecosystem; "intentional" or "passive collaborations", or in "isolated fashion". Therefore, these collaborative networks help companies boost company productivity within the OpenStack. This work is not focused on release synchronization and did not explore the release strategies used by OpenStack. However, the authors were able to show that companies' massive involvement and global engagement increases the complexity of an already challenging task that the central release team faces while coordinating globally distributed teams of contributors/collaborators with different time zones.

A substantial body of work exists on `Dependency Management`, the strategy that we identified within the weekly IRC logs. For example, Constantinou and Mens [40] investigate the dynamics and socio-technical evolution of the software ecosystem by measuring the impact of permanent changes in an ecosystem on the ecosystem as a whole, its developers, and their source code. Results show a significant impact, especially when contributors abandon a project or migrate to another ecosystem.

Decan et al. [11] empirically examine the inter-dependencies of packages in three programming language ecosystems over time and show the extent to which an ecosystem can deteriorate due to package dependency issues.

Bogart et al. [10] show that changes introduced in a package can ripple through an entire ecosystem, which may cause a refactoring of the package. Their results show a design lag in building an ecosystem, especially regarding the policies and infrastructure. Also, the authors claim that building a transparent community can resolve conflicts and change-related costs.

Our research identifies the nine major release synchronization activities used in the thriving OpenStack ecosystem, which (similar to Bogart et al.) involves ecosystem governance and dependencies. OpenStack deals with the ripple-through problem by the use of centralized dependency specifications.

6 Threats to Validity

This section discusses the limitations of our empirical study, following the standard guidelines for empirical studies [55].

Threats to Construct Validity relate to our findings' meaningfulness due to errors in our measurements. Given that the weekly meeting logs and Etherpad documents are explicitly linked and contain a full history of events, we do not suspect any physical errors in this data. Furthermore, since the OpenStack release team's procedures are strict, the meeting logs and Etherpad documents are accurate and precisely reflect what is discussed or worked on during weekly meetings.

We use regular expressions (regex) to search for themes in the archived emails in both the GNOME and Eclipse ecosystem. Using regex could be noisy and subject to erroneous results (false positives). Though we manually verify a random sample of the search results, there is still noise (false positives) in the remaining large number of search results that we could not manually verify. To mitigate this threat, we make sure that we use specific keywords from those false positive matches to strengthen our search query whenever we encounter a false positive match. We add filtered (PIPE or nested search query) that will further eliminate false positives.

We might have missed other relevant high-level activities present in different ecosystems but not found at OpenStack. To mitigate this threat, we consider future works to explore other release synchronization activities.

Threats to Internal Validity relate to alternative explanations that might explain our findings. To reduce the risk of subjectivity in open coding, both authors coded 15% of the logs together, followed by in-depth discussion and resolution of the codes. Besides, an external researcher validated the codes on another OpenStack release, which enabled the authors to calculate the IRR, and this is the typical way in which inter-rater reliability is ensured [28, 29, 30].

Threats to External Validity concern the degree to which findings can be generalized to other OpenStack releases and different ecosystems. We studied two specific releases of the open-source OpenStack ecosystem and an additional release in the middle to validate our findings. However, further studies are needed to explore release synchronization activities in other OpenStack releases. We have also investigated the release synchronization activities in different ecosystems, such as GNOME and Eclipse. We used the activities that we found at OpenStack as our baseline to generalize our findings. However, we might have limited our study from discovering new activities in other ecosystems that were not present in OpenStack. A critical element of such studies is access to release communication or the actual ecosystem release team and individual project members. Where possible, our documented activities try to separate OpenStack-specific from more fundamental concepts.

We also build a persona for a release engineer based on the data we got from the release team members during the interview and partly from their activities over the IRC chat logs. Thus, we try to generalize the data as much as possible (one-size fit all concept [56]) to represent the weekly activities of a typical release engineer at OpenStack. We might have omitted a vital value or characteristic of a release engineer at OpenStack or over-generalized their actual values. To mitigate this threat, we build the persona on the shared data that the release team members share and their average age.

Threats to Reliability Validity. To mitigate such threats, we provide access to the repositories and tools used in this study [37].

7 Conclusion

Software ecosystems are more than the sum of their parts, thanks to the socio-technical dependencies between otherwise independent projects. However, such autonomy comes at a price for the end-user since it becomes challenging to determine the core set of projects (and their versions) that should be installed. Thus, many large open-source ecosystems such as Eclipse, GNOME, OpenStack, etc., overcome these challenges by synchronizing their projects' releases.

This paper empirically studied OpenStack's federated release strategy by analyzing weekly IRC meeting logs for one year (two release cycles). This analysis enabled us to identify ten major release synchronization activities, which we cataloged and documented. To validate our findings, we interviewed eight experts from the OpenStack ecosystem, both the release and project teams.

In particular, given OpenStack's recent dismissal of the "cycle with development milestones" release model, there seems to be ample need to evaluate and improve existing ecosystem release models. Similarly, loosely coupled yet significant interaction between the centralized release team and the individual projects is an ongoing struggle. One release team member wondered. "Are we providing value and helpful information to people and then if we're providing some value, but they could use other information what that might be?" Perhaps inspiration could be found from the concept of social debt in software architecture, where Tamburri et al. discussed the role of software architects as community builders [57]. The challenges regarding Automation Management (A7) are more related to project and risk management, e.g., determining the value of developing new tools or migrating a tool to a more standard technology (technical debt).

Moreover, our findings provide several implications for advising practitioners and academics on the ten release synchronization activities and empirical evidence of why multiple release strategies co-exist in a complex ecosystem. Further, we build a persona for a release engineer, based on common characteristics and values, which practitioners and ecosystem managers could use to hire the right resources into the central release team. Besides, we extracted the release teams' mailing lists archives of GNOME and Eclipse to find patterns of the studied release synchronization activities. We found that we could generalize our findings to other ecosystems that produce federated releases.

Finally, we found the unique activity is Deliverable Consolidation (A8) since this activity captures the essence of release synchronization in ecosystems: integrating artifacts of multiple projects into one central artifact. Hence, we hope to explore this activity in more depth in other ecosystems. The scientific community should also consider an in-depth analysis of how these release synchronization activities happen across different open source communities and how they relate to the **quality** of software deliverables. Studies should also be done to investigate other release synchronization activities in different ecosystems.

References

1. K. Manikas and K. M. Hansen, "Software ecosystems – a systematic literature review," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1294 – 1306, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016412121200338X
2. T. Mens, B. Adams, and J. Marsan, "Towards an interdisciplinary, socio-technical analysis of software ecosystem health," *arXiv preprint arXiv:1711.04532*, 2017.
3. S. Jansen and M. A. Cusumano, "Defining software ecosystems: a survey of software platforms and business network governance," *Software ecosystems: analyzing and managing business networks in the software industry*, vol. 13, 2013.
4. D. M. German, B. Adams, and A. E. Hassan, "The evolution of the r software ecosystem," in *2013 17th European Conference on Software Maintenance and Reengineering*, March 2013, pp. 243–252.
5. K. Plakidas, S. Stevanetic, D. Schall, T. B. Ionescu, and U. Zdun, "How do software ecosystems evolve? a quantitative assessment of the r ecosystem." in *Proceedings of the 20th International Systems and Software Product Line Conference*, ser. SPLC '16. New York, NY, USA: ACM, 2016, pp. 89–98. [Online]. Available: http://doi.acm.org/10.1145/2934466.2934488
6. D. M. Iansiti and G. L. Richards, "The information technology ecosystem: Structure, health, and performance," *The Antitrust Bulletin*, vol. 51, no. 1, pp. 77–110, 2006. [Online]. Available: https://doi.org/10.1177/0003603X0605100104
7. T. Mens and M. Goeminne, "Analysing the evolution of social aspects of open source software ecosystems." in *IWSECO@ ICSOB*, 2011, pp. 1–14.

8. C. R. de Souza, F. Figueira Filho, M. Miranda, R. P. Ferreira, C. Treude, and L. Singer, "The social side of software platform ecosystems," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16. New York, NY, USA: ACM, 2016, pp. 3204–3214. [Online]. Available: http://doi.acm.org/10.1145/2858036.2858431
9. B. Adams, R. Kavanagh, A. E. Hassan, and D. M. German, "An empirical study of integration activities in distributions of open source software," *Empirical Software Engineering*, vol. 21, no. 3, pp. 960–1001, Jun 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9371-y
10. C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: Cost negotiation and community values in three software ecosystems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950325
11. A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in oss packaging ecosystems," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 2–12.
12. R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, May 2017, pp. 102–112.
13. M. Gómez, B. Adams, W. Maalej, M. Monperrus, and R. Rouvoy, "App store 2.0: From crowdsourced information to actionable feedback in mobile ecosystems," *IEEE Software*, vol. 34, no. 2, pp. 81–89, Mar 2017.
14. C. Steglich, S. Marczak, C. R. B. de Souza, L. P. Guerra, L. H. Mosmann, F. F. Filho, and M. Perin, "Social aspects and how they influence mseco developers," in *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering*, ser. CHASE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 99–106. [Online]. Available: https://doi.org/10.1109/CHASE.2019.00032
15. R. Binns, U. Lyngs, M. Van Kleek, J. Zhao, T. Libert, and N. Shadbolt, "Third party tracking in the mobile ecosystem," in *Proceedings of the 10th ACM Conference on Web Science*. ACM, 2018, pp. 23–31.
16. B. Adams and S. McIntosh, "Modern release engineering in a nutshell – why researchers should care," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, March 2016, pp. 78–90.
17. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.
18. F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality? an empirical case study of mozilla firefox," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, June 2012, pp. 179–188.
19. A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin, "Synthesizing continuous deployment practices used in software development," in *Proceedings of the 2015 Agile Conference*, ser. AGILE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/Agile.2015.12
20. B. Adams, R. Kavanagh, A. E. Hassan, and D. M. German, "An empirical study of integration activities in distributions of open source software," *Empirical Software Engineering*, vol. 21, no. 3, pp. 960–1001, 2016.
21. M. Nayebi, B. Adams, and G. Ruhe, "Release practices in mobile apps –- users and developers perception," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, March 2016,

pp. 552–562.
22. J. A. Teixeira and H. Karsten, "Managing to release early, often and on time in the openstack software ecosystem," *Journal of Internet Services and Applications*, vol. 10, no. 1, p. 7, Apr 2019. [Online]. Available: https://doi.org/10.1186/s13174-019-0105-z
23. E. Shihab, Z. M. Jiang, and A. E. Hassan, "On the use of internet relay chat (irc) meetings by developers of the gnome gtk+ project," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, May 2009, pp. 107–110.
24. D. M. German, "The gnome project: a case study of open source, global software development," *Software Process: Improvement and Practice*, vol. 8, no. 4, pp. 201–215, 2004. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spip.189
25. F. G. Longoria, "3 reasons openstack may cost you less than aws," www.forbes.com/, Tech. Rep., Oct 24, 2016. [Online]. Available: https://www.forbes.com/sites/moorinsights/2016/10/24/3-reasons-why-an-openstack-private-cloud-may-cost-you-less-than-amazon-web-services-aws/#4634654348da
26. A. Strauss and J. Corbin, *Basics of qualitative research*. Sage publications, 1990.
27. M. C. Hoepfl *et al.*, "Choosing qualitative research: A primer for technology education researchers," *Volume 9 Issue 1 (fall 1997)*, 1997.
28. J. Saldaña, *The coding manual for qualitative researchers*. Sage, 2015.
29. C. Bird, "Interviews," in *Perspectives on Data Science for Software Engineering*. Morgan Kaufmann, 2016.
30. J. Meinicke, C.-P. Wong, B. Vasilescu, and C. Kästner, "Exploring differences and commonalities between feature flags and configuration options," in *Proc. Int'l Conf. Software Engineering–Software Engineering in Practice (ICSE-SEIP). ACM*, 2020.
31. N. Shrestha, C. Botta, T. Barik, and C. Parnin, "Here we go again: Why is it difficult for developers to learn another programming language?" in *Proceedings of the 42nd International Conference on Software Engineering, ICSE*, 2020.
32. A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 164–175.
33. M. W. DiStaso and D. S. Bortree, "Multi-method analysis of transparency in social media practices: Survey, interviews and content analysis," *Public Relations Review*, vol. 38, no. 3, pp. 511 – 514, 2012, public Relations History. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0363811112000069
34. M. Drouhard, N. Chen, J. Suh, R. Kocielnik, V. Peña-Araya, K. Cen, Xiangyi Zheng, and C. R. Aragon, "Aeonium: Visual analytics to support collaborative qualitative coding," in *2017 IEEE Pacific Visualization Symposium (PacificVis)*, 2017, pp. 220–229.
35. N. McDonald, S. Schoenebeck, and A. Forte, "Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice," *Proc. ACM Hum.-Comput. Interact.*, vol. 3, no. CSCW, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3359174
36. Dedoose, "Dedoose version 8.0.35, web application for managing, analyzing, and presenting qualitative and mixed method research data (2018). los angeles, ca: Sociocultural research consultants, llc," https://www.dedoose.com/, 2018.
37. EMSE-D-19-00279, "Replication package with resources for the qualitative and quantitative study on release synchronization in the software ecosystem." dec 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4304682
38. E. Laukkanen, M. Paasivaara, J. Itkonen, C. Lassenius, and T. Arvonen, "Towards continuous delivery by reducing the feature freeze period: A case study," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engi-*

neering in Practice Track (ICSE-SEIP), May 2017, pp. 23–32.
39. J. Bosch, “Maturity and evolution in software product lines: Approaches, artefacts and organization,” in *Software Product Lines*, G. J. Chastek, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 257–271.
40. E. Constantinou and T. Mens, “Socio-technical evolution of the ruby ecosystem in github,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 34–44.
41. F. Armstrong, F. Khomh, and B. Adams, “Broadcast vs. unicast review technology: Does it matter?” in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 219–229.
42. M. V. Mäntylä, B. Adams, F. Khomh, E. Engström, and K. Petersen, “On rapid releases and software testing: a case study and a semi-systematic literature review,” *Empirical Software Engineering*, vol. 20, no. 5, pp. 1384–1425, 2015.
43. A. Dyck, R. Penners, and H. Lichter, “Towards definitions for release engineering and devops,” in *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, 2015, pp. 3–3.
44. J. Grudin and J. Pruitt, “Personas, participatory design and product development: An infrastructure for engagement,” in *Proc. PDC*, vol. 2, 2002.
45. D. Ford, T. Zimmermann, C. Bird, and N. Nagappan, “Characterizing software engineering work with personas based on knowledge worker actions,” in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017, pp. 394–403.
46. X. Zhang, H.-F. Brown, and A. Shankar, “Data-driven personas: Constructing archetypal users with clickstreams and user telemetry,” in *Proceedings of the 2016 CHI conference on human factors in computing systems*, 2016, pp. 5350–5359.
47. S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona, “Perceval: software project data at your will,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 1–4.
48. S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, and A. Grafberger, “Automating large-scale data quality verification,” *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1781–1794, 2018.
49. F. Liu and Y. Deng, “A fast algorithm for network forecasting time series,” *IEEE Access*, vol. 7, pp. 102 554–102 560, 2019.
50. H. Nguyen and C. K. Hansen, “Short-term electricity load forecasting with time series analysis,” in *2017 IEEE International Conference on Prognostics and Health Management (ICPHM)*, 2017, pp. 214–221.
51. X. Franch and G. Ruhe, “Software release planning,” in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 894–895. [Online]. Available: https://doi.org/10.1145/2889160.2891051
52. M. T. Rahman and P. C. Rigby, “Release stabilization on linux and chrome,” *IEEE Software*, vol. 32, no. 2, pp. 81–88, Mar.-Apr. 2015. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MS.2015.31
53. G. Poo-Caamaño, L. Singer, E. Knauss, and D. M. German, “Herding cats: A case study of release management in an open collaboration ecosystem,” in *Open Source Systems: Integrating Communities*, K. Crowston, I. Hammouda, B. Lundell, G. Robles, J. Gamalielsson, and J. Lindman, Eds. Cham: Springer International Publishing, 2016, pp. 147–162.

54. Y. Zhang, M. Zhou, K.-J. Stol, J. Wu, and Z. Jin, “How do companies collaborate in open source ecosystems? an empirical study of openstack,” *Association for Computing Machinery, New York, NY, USA, 1196–1208*, p. 1196–1208, 2020. [Online]. Available: https://doi.org/10.1145/3377811.3380376
55. R. K. Yin, *Case study research: Design and methods*. Sage publications, 2013. [Online]. Available: https://doi.org/10.1080/09500790.2011.582317
56. B. M. DiCosola III and G. Neff, “Using social comparisons to facilitate healthier choices in online grocery shopping contexts,” in *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–8. [Online]. Available: https://doi.org/10.1145/3334480.3382877
57. D. A. Tamburri, R. Kazman, and H. Fahimi, “The architect’s role in community shepherding,” *IEEE Software*, vol. 33, no. 6, pp. 70–79, Nov.-Dec. 2016. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MS.2016.144