Journal of Internet Services and Applications

RESEARCH Open Access

Managing to release early, often and on time in the OpenStack software ecosystem

José Apolinário Teixeira* and Helena Karsten

Abstract

The dictum of "Release early, release often." by Eric Raymond as the Linux modus operandi highlights the importance of release management in open source software development. However, there are very few empirical studies addressing release management in this context. It is already known that most open source software communities adopt either a feature-based or time-based release strategy. Both have their own advantages and disadvantages that are also context-specific. Recent research reports that many prominent open source software projects have overcome a number of recurrent problems by moving from feature-based to time-based release strategies. In this longitudinal case study, we address the release management practices of OpenStack, a large scale open source project developing cloud computing technologies. We discuss how the release management practices of OpenStack have evolved in terms of chosen strategy and timeframes with close attention to processes and tools. We discuss the number of practical and managerial issues related to release management within the context of large and complex software ecosystems. Our findings also reveal that multiple release management cycles can co-exist in large and complex software ecosystems such as OpenStack.

Keywords: Open source, OSS, FLOSS, Release management, Release engineering, OpenStack

1 Introduction

The dictum of "Release early, release often" by Eric Raymond as the Linux modus operandi [1, 2] highlights the importance of release management in open source software development (see [3–5]). Across disciplines, release management is acknowledged to be a very complex process that raises many issues among the producers and users of software [6–9]. However, there are very few empirical studies addressing release management in open source software development [5, 10]. This is unfortunate since many lessons can be learned from open source software communities [11–13] because they allow studying the socio-technical aspects of software development freely whilst the proprietary model allows access only to a few scholars.

Given this scarcity of empirical work on release management in the context of open source software [5, 10], we address how a particularly large, complex and highly networked open source software ecosystem implemented and refined a time-based release strategy. Taking the case of OpenStack, a fast growing cloud computing platform that is increasingly attracting scholarly attention (e.g., [14–18]), we explore how a time-based release management strategy was implemented in practice by looking at the overall release management process *per se* as well as the infrastructural tools that support it.

By following OpenStack since its inception at the National Aeronautics and Space Administration (NASA)[1], we narrate the evolution of release management at OpenStack. By investigating this 'moving target', we found out that the cross-project release management team relies upon freezes to encourage developers to change their production focus from the development of components to the overall upstream integration and stabilization of components as a whole. After many refinements, OpenStack runs now a time-based release management cycle that is quite liberal (i.e., open to changes and flexible to adaptation). In particular situations, the different sub-project teams across the community are allowed to work around the 'default' six months release cycle. As the project grew, different release cycles started to co-exist across the various OpenStack sub-projects. Release

*Correspondence: jose.teixeira@abo.fi
Åbo Akademi University, Turku, Finland

management started depending heavily on many software tools partially automating the release management process (e.g., tools for version control, revision control, continuous upstream integration, continuous upstream testing, and configuration management). Besides its acknowledged benefits, the implementation of a liberal time-based release strategy is a challenging cooperative task interweaving people with processes, technology, and organizations.

2 Prior related work

Across disciplines, but predominantly in software engineering, plenty of research addressed issues of release management (see [9, 19–24] among others). Whilst we acknowledge and value such research, we note that the studies were mostly conducted in single firms that released proprietary software; a sharp contrast with our case where multiple firms release open source software (i.e., an open source software ecosystem).

On that regard, a framework for analyzing openness in the context of digital platforms and ecosystems recently proposed by Teixeira (2015) [25] raises issues of governance, transparency, market and intellectual property that can be used to juxtapose release management on the single firm releasing proprietary software vs. the network of firms releasing open source software:

- *Governance*. There is a sharp contrast in terms of inclusiveness and control across these two settings. While a single firm releasing proprietary software has more control over its development processes, the network of firms needs to accommodate multiple and often conflicting agendas. Release management in open source software ecosystems needs to accommodate that participants can have different and competing business models and thereby different motivations to engage and contribute. Furthermore, release management in open source projects is more exposed to different mental models, different corporate and individual cultures in a setting or irrevocable openness [26]. In addition, open source ecosystems also tend to be more inclusive to third-party contributors (e.g., students, academics or users among others that do not need a certain organizational affiliation or license to contribute). The power and the influence on deciding what is and what is not released are shared across multiple and heterogeneous participants [16].
- *Transparency*. While it is common for the single firm releasing proprietary software to hide information about the software being released (e.g., the source code, the bug tracking information), the same is not customary in open source software ecosystems where access and transparency are required for the community to function. It is also worth remarking that while most open source software is released on the Internet (i.e. to a repository or a website where it can be consumed by others), much proprietary software is only considered released once deployed at the customers' production environment [24].
- *Market*. While a single firm releasing proprietary software has more control over the commercialization of the software and its complements, in the case of open source software ecosystems, the value exploration around a component can not be fully controlled by a single firm. Multiple firms can compete for the value around the 'common' software being released. In the case of OpenStack, when deciding whether a certain open source component is released or not, a number of issues can be raised, such as 'Will the component compete with some of our proprietary offerings (e.g., plug-ins)?', 'Will the component allow us to win some consulting and deployment contracts?', 'Will the component increase the demand for our hardware?'or 'Will the component increase the demand for our hosting infrastructure?' among other market-related issues that can impact release management.
- *Intellectual property rights*. Releasing software can create *prior art* in terms of intellectual property rights. Single firms releasing proprietary software tend to deal more with intellectual property protection issues. In open software ecosystems, there might be intellectual property issues as well, but they often need to be resolved across multiple participants. Furthermore, by releasing under an open source software license, the developers or firms are giving up rights that are automatically granted by law. Note that it is not uncommon for firms to submit related patents prior to releasing a certain software component. This aspect is more relevant in markets where software can be patentable (USA, Japan, and South Korea) [27]. From the point of view of brands and trademarks, there is also a sharp contrast. While a single firm often has control over the trademark of a given software product, in the case of open source software such control is often multi-lateral and negotiated across the different individuals and organizations that contribute to the joint development of open source software. After all, the quality of a given release can affect the value of the brand and the trademark associated with the software being released.

Regarding release management within the specific context of open source software, it is known that release management affects both producers and users of software. On the producer side, prior research suggests that

community activity increases when the scheduled release date gets closer [28]. On the user side, new releases result in spikes of downloads [29]. Library projects where new code libraries are developed and client projects where the libraries are reused organize release management in a different way to accomodate their technical dependencies. On that regard, recent results suggest that client projects are quicker to update libraries with a rapid release cycle compared to actual library projects with a longer release cycle [30]. The same study also suggests that client projects are more likely to adopt the latest version of libraries with shorter release cycles. As noted in the early work of Michlmayr [31] focusing on release management in open source software, release management is concerned with the delivery of products to end users. It is therefore not surprising that recently some have seen release management as a process that supports value co-creation among suppliers and consumers of software [6, 23].

As pointed out by three recent doctoral dissertations addressing release management in the context of open source software [31–33], most open source software communities adopt either a feature-based or a time-based release strategy. Many prominent open source software projects start with sporadic releases in which developers announce the newly developed features[2]. However, as many of these projects grew in size and complexity, they started adopting time-based release strategies[3]. An early empirical study [34] that mined the repository of a project while it adopted a time-based release strategy (i.e., the evolution of an e-mail client), reported that the adoption of a time-based release cycle boosted the development in general terms over time in comparison to feature-based release cycles. There are many problems associated with feature-based strategies. For example, if critical features are not ready, they block the overall release. Another example is when developers work on the features they are interested in, coordinating their activities can be challenging [35]. A recent studies based on interviews with key members of seven prominent volunteer-based open source projects, points out that these problems can be overcome by employing a time-based release strategy [5, 36].

Time-based release strategies encompass meeting a schedule, an agenda, or a deadline. These can be either strict or liberal. To enforce that software is released on time, the use of freezes (such as code freezes) will set a clear deadline for the software development team. Open source developers have much freedom to manage their own software development efforts, but the use of freezes acts in the opposite way, by constraining the developers. If new features are not implemented before the upcoming freeze, they will not be included in the next release. Consequently, when developers realize this, the development of these features is either canceled, put on hold, or developed separately for future releases.

The freezes that occur before the scheduled time-based release, act as control mechanisms that slowly halt the production of the development core code [13, 37]. In large and complex open source software projects with a modular architecture integrating components with each other, this kind of a freeze forces developers to (1) fix and release the individual components upstream, (2) integrate the different components and test the integrated totality.

According to Fitzgerald, the freeze categories can include [13]:

feature freeze no new functionality can be added, the focus should be on removing defects;

string freeze no messages displayed by the program, such as error messages, can be changed. This allows translating as many messages as possible before the release;[4]

code freeze a permission is required to make any changes, even to fix bugs.

3 Empirical background

The cloud computing business is dominated by a small number of players (e.g., Amazon, Google and Microsoft). The leading players do not sell cloud infrastructure products. Instead, they provide bundled computing services. If there was no alternatives, all cloud computation would run in hardware and software infrastructures controlled by very few players with increased customer lock-in [16].

Competing with the providers of these services, the leading product alternatives are not commercial. Instead, they are four open source projects: OpenStack, CloudStack, OpenNebula, and Eucalyptus. While the commercial cloud computing services are developed and tightly controlled by a single organization, the open source products are more inclusive and networked where multiple firms participate in their development and where multiple firms attempt to capture value from them.

OpenStack is an open source software cloud computing infrastructure capable of handling big data. It is primarily deployed as an Service (IaaS) solution. It started in 2010 as a joint project of Rackspace, an established web hosting company, and NASA, the U.S. governmental agency responsible for the civilian space program, aeronautics and aerospace research. As it evolved, the project attracted much attention from the industry. By the end of 2017, OpenStack counted with more than 82,000 contributors and 670 supporting companies. Furthermore, more than 20 million lines of code were contributed by developers in 187 countries[5].

Both private companies (e.g., AT&T, AMD, Canonical, Cisco, Ericsson, HP, IBM, Intel, VMware, Citrix, and NEC, among many others) and research-intensive organizations (e.g., NASA, CERN, Johns Hopkins University, Instituto de Telecomunicações, Universidade Federal de Campina Grande, and Kungliga Tekniska Högskolan, among others) work together with independent, non-affiliated developers in a scenario of pooled R&D. They work in the open source way, that is, emphasizing development transparency while giving up intellectual property rights. Paradoxically, even though OpenStack emphasizes collaboration in the joint development of a large open source ecosystem, many participating firms also compete with each other. Among others, there is competition among providers of public cloud services based on OpenStack (e.g., HP, Canonical, and Rackspace), among providers of specialized hardware complementing OpenStack (e.g., HP, IBM, and Nebula), and among providers of complementary commercial software plug-ins that complement OpenStack (e.g., VMware, Citrix, and Cisco) (see [16, 38] for related research addressing cooperation among competitors within OpenStack).

We decided to address OpenStack due to its perceived novelty, its highly inter-networked nature (i.e., an ecosystem involving many firms and individual contributors), its heterogeneity (i.e., an ecosystem involving both startups and high-tech corporate giants), its market size ($1.7bn, by 2016[6]), its complexity (i.e., involving different programming languages, different operating systems, different hardware configurations) and size (20 million lines of code contributed by more than 82,000 developers).

From the early beginnings of OpenStack, the project adopted a liberal six-month, time-based release cycle with frequent development milestones that raised much discussion among its developers. We found it an interesting case to study release management within the overlap of open source software, software ecosystems, and complex software systems.

4 Methodological design

This empirical case study was guided by the broad research question on "How OpenStack implemented a time-based release strategy?" A particular emphasis was given to the release management process *per se* as well as to the infrastructural tools supporting it. Tools are focal as the OpenStack community attempts to automate the release management process as much as possible[7].

Our efforts were built on top of publicly available and naturally occurring archival data derived from the OpenStack project. These data are not a consequence of our own actions as researchers, but are created and maintained by the OpenStack community in their own pursuits in developing a cloud computing infrastructure. We took into account many methodological notes in case study research that legitimate the use of archival data when studying a case [39–43].

We started by 'digesting' many websites officially related to OpenStack (e.g., https://www.openstack.org/, https://wiki.openstack.org and http://docs.openstack.org/), expanding later to other websites. When selecting the initial sources (i.e., departure points), we took into consideration key guidelines on how to conduct qualitative empirical research online [44, 45]. From the initial sources, we followed several links to collect further information. The links often led to blogs maintained by organizations and individuals that recurrently contribute to OpenStack. Relevant data was meticulously organized in a database for later analysis [46, pp 94-98].

From our initial screening of qualitative data, we were able to: (1) make sense of the industrial background in which Openstack is embedded, (2) make sense of the complex software development processes that steer the project evolution, (3) survey complex inter-organizational arrangements within the project, and (4) understand the role of many of the software tools that support its software development processes. After getting familiar with many social-technical issues within OpenStack, we analyzed the collected data with the lenses of extant knowledge in release management and open source software.

Given the scarcity of theoretical and empirical knowledge addressing release management in open source software [5, 10], we explored and narrate the evolution of release management in practice. Our rich narrative on how OpenStack implemented a six-month, time-based release cycle with frequent development milestones can increase our ability to understand and explain release management within the context of complex open source software ecosystems.

To enhance the validity of our narrative, we asked early four OpenStack developers (two of them with release management responsibilities) to read and comment our Sections 5 and 6 in advance. One of our informants joined the project in November 2010 with the responsibility of organizing release management at OpenStack and reports since then to the technical committee of the OpenStack foundation. Another of our informants joined OpenStack in 2014 and currently leads a subproject and recurrently attends cross-project team meetings dealing with release management. While the first one has a more managerial view on release management at OpenStack, the second one sees release management from the complementary developers' perspective. In this way we reduced possible misinterpretations of the collected data.

Our description of release management in the particular case of OpenStack, that implemented a time-based release management strategy is organized around a dual view that aggregates two different aspects of release management. First, a view on release management as a complex and evolving inter-organizational socio-technical process; and second, an infrastructural view on the tools that support it.

5 Results: a processual view

Our results aim at contributing to a better understanding of release management within complex open source software ecosystems. Release management can take many distinct forms and many lessons can be learned from how it is done in an open source software community [13].

Given that release management at OpenStack evolves dynamically as a 'moving target', we organized our narrative across three phases of OpenStack. First, we briefly describe its early days as an internal project of NASA. Then, we look at its bootstrap as an open source software project largely steered by Rackspace (at that time supplying NASA with cloud computing services). After that, we concentrate our attention on describing release management at OpenStack at a more recent and mature phase (i.e., release management of OpenStack as by the end of the year 2017). We provide a narrative on the evolution of release management at OpenStack while purposively focusing on its more recent and mature phases from which more lessons can be learned.

5.1 Early days at NASA

The OpenStack project was officially announced on July 21, 2010 at the Open Source Convention (OSCON) (a business and technical oriented convention organized by O'Reilly Media that annually gathers many open source contributors across different cities of USA)[1]. However, the technology behind OpenStack started much earlier at NASA (a well-known space and aeronautics research agency) and Rackspace (a popular provider of hosting services).

At that time, Anso Labs had published the beta code for *Nova*, a cloud computing fabric controller implemented in the C, C++ and Python programming languages. Anso Labs was later acquired by Rackspace on February 9, 2011 and since then *Nova* remains a core component of OpenStack. Meanwhile, Rackspace also wanted to rewrite the infrastructure code running its offering for cloud servers, and open sourced the existing cloud files. This open sourcing of code, previously held in-house, led to the creation of *Swift* that is OpenStack's scalable redundant object storage system. As an open source project, *Swift* was able to address the exceptionally demanding storage needs of NASA. The first release of OpenStack had then two main components: *Nova* and *Swift*. This is how OpenStack started (see [47] for a more detailed narrative on how OpenStack started at the hands of NASA, Anso Labs, Rackspace, and others).

These initial software components were not developed in the open source arena but in-house. The software was released once new features were ready. This is in a big contrast with OpenStack today, in terms of release strategy and the overall development transparency. Nowadays, OpenStack releases trail a six-month release schedule and tarballs[8] are available for every single code commit. In the particular case of *Nova* at NASA, its release management was influenced by procedural requirements at NASA: all software developed in and for NASA should follow comprehensively documented procedures such as Release Management (SWE-085), SW Development-Management Plan (SWE-102), and Release of NASA Software (NPR 2210) among many others[9].

Nowadays, NASA has its own documented procedures on how to release open source software. So far, NASA has released more than 60 software projects under the NASA Open Source Agreement (NOSA), a non-permissive license created and approved by the Open Source Initiative (OSI) in 2013. Note, however, that in the early days of OpenStack at NASA, the release of software to the open source community was uncommon[10].

5.2 Bootstrap as an open source project

The mission of OpenStack, in the announcement in July 2010, was to "to produce the ubiquitous Open Source Cloud Computing platform that will meet the needs of public and private clouds regardless of size, by being simple to implement and massively scalable." The first OpenStack design summit was held in Austin, Texas, USA during July 2010[1]. The announcement highlighted prior work made in NASA. Citrix, a long standing partner of Rackspace on virtualization technologies, joined the project. The project kept growing with more and more partners (e.g., Intel, Cisco, Nebula, CloudScale, HP, Mirantis, Canonical, VMware, RedHat, and IBM among others) and the OpenStack foundation was founded as a non-profit corporate entity, established in September 2012, to manage and promote OpenStack and its community. (see [16, 47] for more details on how OpenStack open source project grew over time). For protecting the brand, OpenStack holds the trademark and hires staff that deal directly with release management from the development side (engineering, design, infrastructure, ecosystem management, etc.) as well as the marketing side (events, communications, launching, promoting, etc.).

As OpenStack was growing in terms of community, code, and complexity. Thierry Carrez, an engineer with much experience in Linux distributions (Ubuntu and

Gentoo) was hired by Rackspace to do release management at OpenStack. As noted in the following quotation, and in congruence with prior research on why open source software projects should adopt time-based releases [5], a time-based release strategy sets a cadence, a rhythm, a pulse of contributions across the project. Moreover, the setting of release dates allows developers to shift their focus from feature development to bugfixing – all towards a better quality of the software being released.

> 'The most obvious value of release cycles is to help producing a release. It allows us to shift focus from feature development to release-critical bugfixing, which results in better quality in the end release. But to me, the most important value of release cycles is that they create a common rhythm of contribution, a common pulse, which is essential for our virtual and global community to feel part of the same project' — Thierry Carrez , 1 July 2013[11].

The project started with a 3-month release cycle, but at the third OpenStack Design Summit in Santa Clara, April 26-29, 2011, they decided to switch from a 3-month cycle to a 6-month coordinated release cycle, with more frequent milestone deliveries in the middle[12]. Given the resources of the community at that time, developers had difficulties to land new features in such a short time window. As evidenced by the following quotation, the long time stabilizing and maintaining what was developed so far drove the community towards the longer 6-month release cycle. In this sense, the release management cycle length started by resembling Linux (2–3 months) but evolved to what we can see at other reference open source projects such as Ubuntu, GCC, X.org and GNOME (6 months) [5].

> 'You may remember that we used to have 3-month cycles at the beginning of OpenStack. Currently, it takes us about 4 weeks after we stop adding features to come up with a valid release candidate. In a 6-month release cycle, it's acceptable to be feature-frozen for one month. In a 3-month release cycle, less so. Releasing every 3 months also means maintaining twice as many stable branches. So if more people addressed critical bugs during the rest of the cycle (when we are not feature-frozen) and more people helped with stable branch maintenance and security updates, we could definitely consider going for 3-month cycles. I like to have a Design Summit at the beginning of each cycle (I think it helps us deliver better results), so we'd probably also have to convince the Foundation to pay for twice as many developer events' — Thierry Carrez , 1 July 2013[13].

5.3 Reaching maturity as complex open source project

The first release, code-named 'Austin', appeared four months after the OpenStack announcement at OSCON, with plans to release regular updates of the software every few months. 'Austin' was already a sizable release as it inherited the code base from NASA's Nebula platform as well as the code base from Rackspace's Cloud Files platform. Firms such as Canonical, SUSE, Debian and Red Hat, all with a recognized role in the open software world, were among the first organizations engaging with OpenStack. Citrix, HP, and IBM were among the first high-tech giants that contributed to development of the project.

As OpenStack increased both in size and complexity, the forthcoming releases code-named 'Bexar', 'Cactus', and 'Diablo' came at irregular intervals that ranged from three to five months[14]. As captured by the following quote, the 'Diablo' was the first of many forthcoming releases launched within a six months release cycle.

> "This release marks the first six month release cycle of OpenStack. The next release, Essex, will also be a six month release cycle and development is now officially underway. While Diablo includes over 70 new features, the theme is scalability, availability, and stability." — Devin Carlen, 29 September 2011[15].

OpenStack is so far orchestrated by the Git distributed version control system (aka repository) and the Gerrit revision control system (aka code review tool). The source code of OpenStack is hosted across dozens of repositories[16]. Due to the inherent complexity of a large-scale project developed by dozens of firms and hundred of developers, keeping everything within a single repository would raise issues about when and where bugs are introduced or how to trace longitudinally the development of features. Moreover, by using a multiple repository approach, access control can be customized to each individual repository. New developers need not spend so much time learning the structure of a large source code tree, and small changes across the multiple projects would not bother so much the other projects. Additionally, OpenStack also attempted a modular architecture with various components, where each project team is responsible for managing its own component repository[17]. Some components, such as the OpenStack Compute (aka Nova and the computing fabric controller), are core components on which many other components rely. To be able to integrate with these components, modular designs and much cross-project coordination is required.

> "We started this five-year mission with two projects: Nova (Compute) and Shift (Object Store) and over time, the number of projects in OpenStack grew. Some of this where parts of the existing projects that split out to have their own separate teams and become little more

> modular. Other things were good new ideas that people had that fit within the realm of OpenStack. Like interesting things that you would want to do in or with a cloud. Over time, we built a process around that to deal with the fact that there were so many of these projects coming in." — Sean Dague , 15 May 2015[18]

OpenStack keeps refining its release management process but it is adhering to a six-month release cycle. Each release cycle encompasses planning (1 month), implementation (3 months), and integration (2 months) where most pre-release critical bugs should be fixed. During the earlier release phase, the coding efforts are much driven by discussion and specifications, while in a later release phase (i.e., stabilization of release candidates) the development turns into the bugfixing mode (as reported in other open source projects [5, 28, 34]). At each release, developers start by implementing the discussed and/or specified key features while, by the end of the release, there is a peak of bugfixing activities. To sum up, each release cycle starts in a specification and discussion driven way and ends in a bug tracker oriented way.

The planning stage is at the start of a cycle, just after the previous release. After a period of much stress to make the quality of the previous release acceptable, the community steps back and focuses on what should be done for the next release. This phase usually lasts four weeks and runs in parallel with the OpenStack Design Summit on the third week (in a mixture of virtual and face-to-face collaboration). The community discusses among peers while gathering feedback and comments. In most cases, specification documents are proposed via an infrastructure system[19] that should describe precisely what should be done. Contributors may propose new specs at any moment in the cycle, not just during the planning stage. However, doing so during the planning stage is preferred, so that contributors can benefit from the Design Summit discussion and the elected Project Team Leads (PTLs) can include those features into their cycle roadmap. Once a specification is approved by the corresponding project leadership, implementation is tracked in a feature blueprint[20], where a priority is set and a target milestone is defined, communicating when in the cycle the feature is likely to go live. At this stage, the process reflects the principles of agile methods.

The implementation stage is when contributors actually write the code (or produce documentation and test cases, among other software-related artifacts) mapping the defined blueprints. This phase is characterized by milestone iterations (once again a characteristic of agile software development methods). Once developers perceive their work as ready to be proposed for merging into the master branch, it is pushed to OpenStack's Gerrit review system for public review[21]. It is important to remark that in order to be reviewed in time for a milestone, the change should be proposed a few weeks before the targeted milestone publication date. An open source software collaboration platform[22] is used to track blueprints in the implementation stage. In a more open source way and not to discourage contributors, it is worth remarking that not all features have to go through the blueprints tracking: contributors are free to submit any ad hoc patch. Both specifications and blueprints are tools supporting the discussion, design, and progress tracking of the major features in a release. Although the big corporate contributors are naturally more influential in the election of PTLs steering the tracking process, this should not prevent other contributors from pushing code and fixes into OpenStack. Development milestones are tagged directly on the master branch during a two-day window (typically between the Tuesday and the Thursday of a milestone week). At this stage, heavy infrastructure tools that continuously integrate and test the new code play a very important role[23].

At the last development milestone, the OpenStack release management team applies three feature freezes: *FeatureFreeze*, *SoftStringFreeze* and *HardStringFreeze* that gradually constrain the evolution of the code base as described in Table 1. At this point, the project stops accepting new features or other disruptive changes. It concentrates on stabilization, packaging, and translation. The project turns then into a pre-release stage, termed as 'release candidates dance'[24]. Contributors are encouraged to turn most of their attention to testing the result of the development efforts and to fix release-critical bugs. Critical missing features, dubious features, and bugs are documented, filed and prioritized. Contributors are advised to turn their attention to the quality of the software

Table 1 The three feature freezes of OpenStack

Freeze	Description
FeatureFreeze	Project teams are requested to stop merging code adding new features, new dependencies, new configuration options, database schema changes, changes in strings ... all things that make the work of packagers, documenters or testers more difficult.
SoftStringFreeze	After the FeatureFreeze, translators start to translate the strings. To aid their work, any changes to existing strings is avoided, as this will invalidate some of their translation work. New strings are allowed for things like new log messages, as in many cases leaving those strings untranslated is better than not having any message at all.
HardStringFreezee	10 days after the SoftStringFreeze, any string change after RC1 should be discussed with the translation team.

and its documentation. The development becomes mainly bugfixing oriented and a set of norms and tools guide this last product stabilization phase[25]. Between the last milestone and the publication of the first release candidate, contributors are urged to stop adding features and concentrate on bug fixes. Only changes that fix bugs and do not introduce new features should be allowed to enter the master branch during this period. Any change proposed for the master branch should at least reference one bug on the bug tracking system. Once all the critical bugs for the release are fixed, OpenStack produces the first release candidate for that project (named RC1). Across this last stage, the repository version control system (i.e., Git) plays an important role in alleviating the interruption caused by the freezes. A freeze applies only to the stable branch so that developers can continue their work on other the development branches (i.e., the trunk). New features should be committed to other branches, discussed at the planning stage, and merged into the stable branch at the next implementation stage.

The OpenStack release team is empowered during this last phase. It creates a `stable/*` branch from the current state of the *master branch* and uses Access Control List (ACL) mechanisms to introduce any new release-critical fixes discovered by the release day. In other words, further changes at this stage require permission from the release team. In the words of OpenStack, they will be treated as feature Feature Freeze Exceptions (FFEs). Between the RC1 and the final release, OpenStack looks for regression and integration issues. RC1 may be used *as is* for the final release unless new release-critical issues are found that warrant an RC respinning. If this happens, a new milestone will be open (RC2), with bugs attached to it. Those RC bug fixes need to be merged in the *master branch* before they are allowed to land in the `stable/*` branch. Once all release-critical bugs are fixed, the new RC is published. This process is repeated as many times as necessary before the final release. As the final release date gets closer, to avoid introducing last-minute regressions, the release team limits the number of changes and their impact: only extremely critical and non-invasive bug fixes can get merged. All the other bugs are documented as known issues in the Release Notes instead.

On the release day, the last published Release Candidate (RC) of each integrated project is collected and the result is published collectively as the OpenStack release for this cycle. OpenStack should by then be stable enough for real industrial deployments. Once the version is released, a new cycle will commence within OpenStack; the *master branch* switches to the next development cycle, new features can be merged freely, and the process starts again. After the release and a period of much stress that required much coordination, most of the community shifts again to the planning stage and many will attend the Design Summit. A new branch has been opened already to accommodate new developments. Even so, the launched release needs to be maintained and further stabilized until its End of Life (EOL), when it is no longer officially supported by the community. OpenStack might release bugfix updates on top of previously announced releases with fixed bugs and resolved security issues, actions that might distract developers working on newer items.

The overall release management process, as illustrated in Fig. 1, follows a plan, implement, freeze, stabilize and launch cycle between releases. Each release is then re-stabilized with *a posteriori* release updates to fix bugs and security issues. Nevertheless, the process described so far is just the most recurrent pattern within OpenStack, the default *modus operandi*. The described process is actually quite open and liberal. It acts as a recommendation for the different teams so that whatever is developed is then later more smoothly integrated, stabilized and released in a coordinated fashion.

Since October 2016 (affecting the 'Newton' release), OpenStack actually recommends its project teams to choose from four different release management models: *Common cycle with development milestones*, *Common cycle with intermediary releases*, *Trailing the common cycle* and *Independent release model*. Most of these models follow a common six-month development cycle, some give intermediary releases within the six-months cycle and others are allowed to manage their own release strategy[26].

Common cycle with development milestones The official and default time-based model followed by most teams. It results in a single release at the end of the development cycle and includes three development milestones (as in Fig. 1).

Common cycle with intermediary releases For project teams wanting to do a formal release more often, but still want to coordinate a release at the end of the cycle from which to maintain a stable branch. Recommended for libraries, and for more stable components, which add a limited set of new features and do not plan to go through large architectural changes.

Trailing the common cycle For project teams that rely on the completeness of other components (e.g., packaging, translation, and UI testing) and may not publish their final release at the same time the other projects. For example, teams packaging and deploying OpenStack components need the final releases of many other components to be available before they can run their own final tests. Cycle-trailing project teams are given an extra two weeks after the official release date to request the publication of their

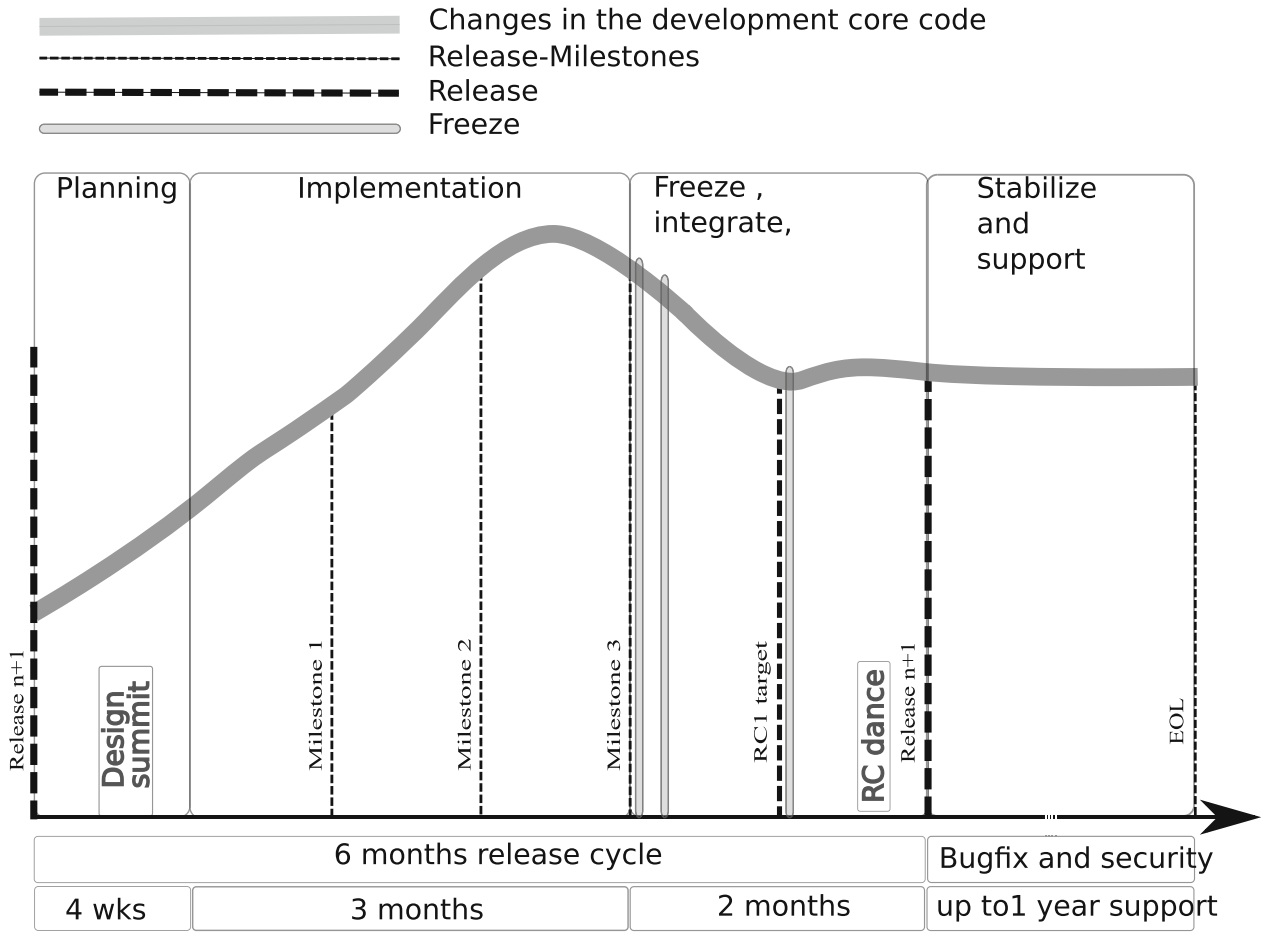

Fig. 1 Overview of the OpenStack standard release cycle

own releases. They may otherwise use intermediary releases or development milestones.

Independent release model For project teams that do not benefit from a coordinated release or from stable branches. They may opt to follow a completely independent release model. Suitable for example for the OpenStack's own infrastructural systems (e.g., the ones supporting upstream testing and integration) as well for components with little dependence on the overall Openstack core architecture.

"We still have a coordinated release at the end of the six months for projects that are willing to adhere to those deadlines and milestones, but the main change is that we will move from managing most of them to refine processes and tools for each project to be able to produce those releases more easily. The development cycle will still be using a six months development cycle, even if some projects might do intermediary releases where it makes sense, but will still organize almost everything under a six months development cycle between design summits." — Thierry Carrez , 15 May 2015[27].

In an attempt to sump up and aggregate key elements of our narrative, the timeline in Fig. 2 highlights key events and turning points on the evolution of release management at OpenStack. From the first days, when the overall development was shaped by the official software development processes institutionalized at NASA, then the spin-off as an open source project with releases at every three months, then the shift towards a more liberal release cycle of six months, and later, after a period of much growth, the co-existence of multiple release models trailing a common six-month release cycle.

6 Results: infrastructural tools

Given that release management at OpenStack relies on a vast tool-chain, we start by addressing tools that are mostly used at the beginning of the release cycle, in discussion, planning and specifying. After that, we cover tools that are more widely used during the implementation and stabilization of new features at the later stage of

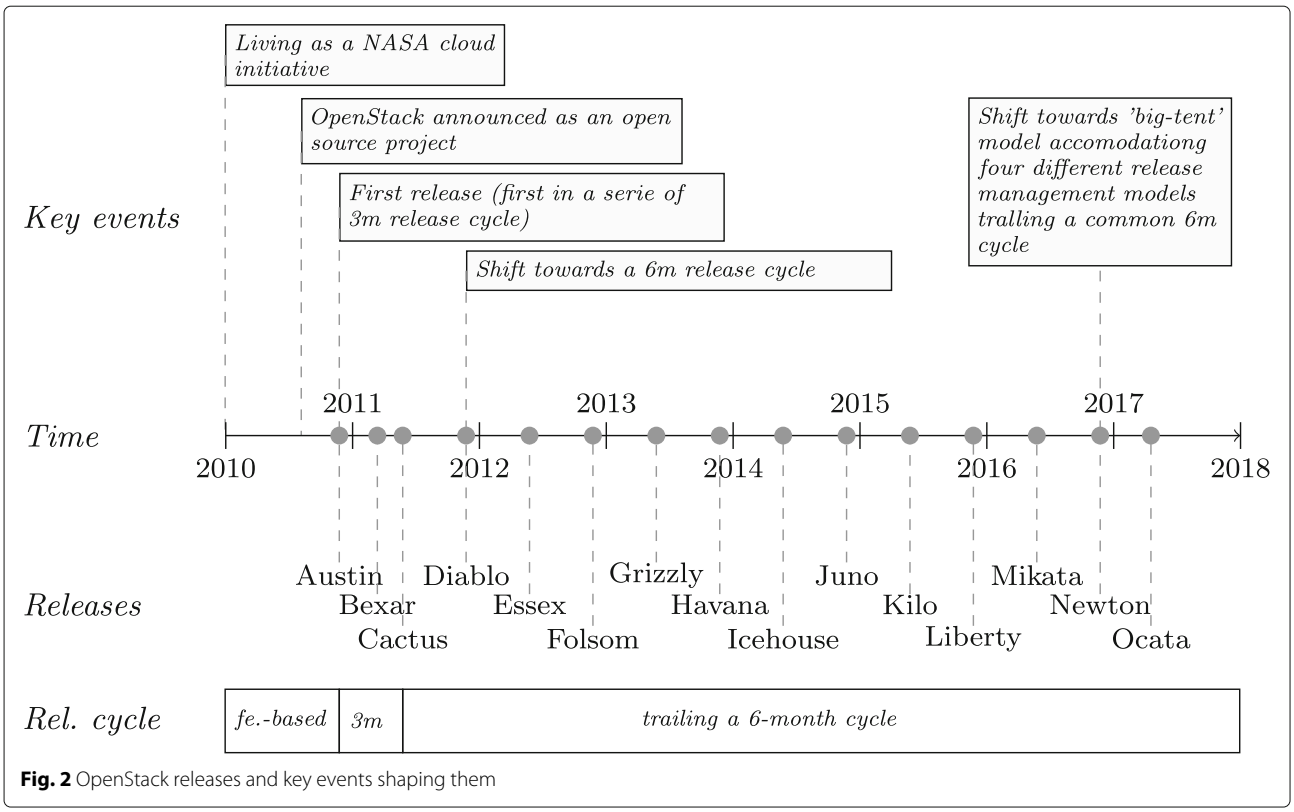

Fig. 2 OpenStack releases and key events shaping them

the release cycle. These include tools for version and revision control, reviewing and testing. The closer developers are to the end of the release cycle, the more important is the role of the tools supporting continuous integration, testing, and code-reviewing. We close with infrastructural tools supporting release management, elaborating on the novel OpenStack 'micro-tagging' approach and a new 'release notes manager' tool developed within the community as part of their own release management endeavors.

6.1 Discussion, planning and specifying

Starting a new release cycle emerges with many specifications and discussions. At this planning stage, developers gradually leave the *integration* and *stabilization* mode to enter again into the *planning* and *development* mode. Besides considering what should be done, developers concerned with release management consider what can be done on time to be included in the upcoming release.

As the planning stage partially overlaps with the OpenStack design summit, many of the discussions take place face-to-face. However, for a number of reasons (e.g., the discussions at the summit are fragmented in multiple parallel sessions and many developers are not able to attend the summit at all), developers use a number of infrastructural tools that facilitate the discussion and formalization of what should be developed during the next milestones. Even if the different project teams are free to use their own tools in an ad hoc manner, multiple communication, coordination, and collaboration tools are provided by the OpenStack infrastructure team. Among others, developers rely on *LaunchPad* for blueprints, *Sphinx* for specifications and *StoryBoard* for task tracking. As seen also in other open source projects, communication and information sharing is supported by standard Web, Wiki, IRC and e-mail systems.

According to Poo-Caamano [10], coordination across multiple project teams requires a common infrastructure to facilitate communication to flow between the release team and the projects. Asynchronous channels, such as mailing lists, act as egalitarian media for discussion among a wide range of participants as the messages can be archived and made searchable. Conversely, synchronous channels, such as face-to-face meetings, may be richer, but they are less inclusive as they may exclude participants that cannot attend a physical face-to-face discussion. Furthermore, as English is de facto the *lingua franca* of most open source projects, asynchronous channels might be preferred by developers that may not be as fluent as their native English-speaking team mates [10].

6.2 Orchestrating distributed work, tagging and version control

Managing the release of software at scale requires a good orchestration of its evolving code base. Since the early days of OpenStack, Git[28] plays an important role in

release management at OpenStack. In a highly distributed software development environment with thousands of developers, the branching and merging capabilities of Git are essential for governing what code should be at each branch of the official OpenStack repository. Cloning, fetching and pulling from the official repository maintained by the OpenStack downstream to the developers' local environment, as well as branching, committing and pushing upstream, are common OpenStack development operations that rely on Git.

As other distributed version control systems, Git has the ability to tag specific points in the source code history as being important. Therefore, it is a common procedure to do 'versioning' (ie., naming the release) using Git tags[29]. The use of tags allows both developers and users of OpenStack to track the incremental evolution of the software being developed intuitively. Naming different versions of the software is essential for both developers and users of OpenStack that often need to deal with multiple versions of OpenStack.

As Git tags apply only to a commit and are not branch-aware[30], OpenStack developers encode key release information such as project name, release series, branch and commit hash, within plain YAML[31] text files. Developers produce a large variety of code that is hosted across multiple Git repositories. Given the complexity of the OpenStack, such repositories are highly interdependent and tightly integrated. For example, a repository might host a library used by multiple teams in multiple components, or a core Application Programming Interface (API) might be 'called' by the many distinct services that are implemented in OpenStack. Therefore, once developers formally release their work, their deliverable may span across multiple repositories. At the operational level, this means that developers might need to encode multiple (project, commit hash) tuples in a YAML file. The more repositories developers change, the higher number of YAML files needs to be delivered or the more complex a single YAML file will be.

Enriching this description of release management in practice at OpenStack, the following illustrative YAML file 1, elucidates how the *neutron* project team delivered the version *10.0.0.0rc1* from the *ocata* stable release series that pointed out to the commit *4ae6790d82542738edbb531a829b60ff8a44a3fe* from the *neutron* Git repository and the commit *8cfa2de66becce06f3e11bbab7562b11649e54c9* from the *neutron-fwaas* Git repository[32]. When the different project teams encode this information in their deliverables, the release management team is able to verify better the completeness and consistency of the overall contributions that are to be released. Furthermore, as this information is provided openly[33] in a standardized and machine-readable way, it enables a high level of automation of the release management activities.

The ability of Git to mark specific points in the source code history as tags is accompanied with support for annotating and signing them using GnuPG[34] cryptographic keys. On the one hand, the tagging functionality of Git is often used to mark release points (v1.0, and so on), while on the other hand, the annotating and signing functionalities allow developers to add securely their name, email and tagging message (often the version). The annotating and signing functionalities provide security in the sense of integrity checking, where the other developers can check if the tag and the corresponding code were really issued by a trusted developer. If the Git branching functionality allows the developers to deal with *n* different versions of the repository at the same time, the tagging functionality allows developers to mark a point in time in the repository that is not aware of its branch. It is a good practice for developers to tag and sign to mark a released version and, if post-release bug fixing actions are required, to create a bug-fixing branch at the tag[35]. In the particular case of OpenStack, the use of tags gained

```
---
launchpad: neutron
team: neutron
release-notes: https://docs.openstack.org/releasenotes/neutron/ocata.html
type: service
release-model: cycle-with-milestones
branches:
  - name: stable/ocata
    location: 10.0.0.0rc1
releases:
  - version: 10.0.0.0rc1
    projects:
      - repo: openstack/neutron
        hash: 4ae6790d82542738edbb531a829b60ff8a44a3fe
      - repo: openstack/neutron-fwaas
        hash: 8cfa2de66becce06f3e11bbab7562b11649e54c9
    highlights: |-
      * ConntrackNetlink driver to delete conntrack entries
      * XenAPI: Support daemon mode for rootwra
```

Illustrative YAML file 1 - Release information formalized in a file.

much importance for release management. In the beginning of OpenStack, it was a selection of core and mature repositories at a given time that dictated what was in the official OpenStack release. This is called the integrated release model. Since the OpenStack Liberty release (October 2015), tags signed by the Technical Committee (TC) dictate what is the official OpenStack release. This is called the big tent release model. In other words, since the Liberty release, OpenStack software releases are not seen as a selection of core repositories, but as code tagged by the TC across all the repositories of OpenStack. It is important to note that only software officially released by OpenStack can use the OpenStack trademarks for example for marketing of the software using the OpenStack name or logo[36].

6.3 Reviewing

Release management includes deciding if a certain piece of code is ready to be released or not. In other words, stakeholders in the release management must ensure whether the quality of certain software is at an acceptable level to be released or not. Therefore, release management and peer review are inseparable processes [20, 22, 48, 49]. Many successful open source software projects employ formal code review activities prior to the release[37]. Core reviews are effective means for quality assurance, knowledge dissemination, social relationship building, achieving better designs, and ensuring maintainable code in long term [50]. In the OpenStack case, peer review is handled by Gerrit[38], a tool that hides many of the complexities of reviewing code by directly using Git commands under a web-based tool designed solely for the purpose of supporting code review activities. As in many other open source projects, the submitted code is reviewed before it is accepted into the official code base prior to the release announcements. While in many other projects developers submit more atomic commits or patches for review, in the case of OpenStack, developers submit Git branches that are more bulky units, to be reviewed. To do so, OpenStack developed and recommends using the Git review tool for submitting code. This tool is invoked as a Git subcommand and it handles all the details of working with Gerrit. The process is quite straightforward: developers implement new features in their own local Git branch, commit their changes locally and then simply invoke the *git review* subcommand that will submit the 'patchset' to Gerrit. Developers will then receive an acknowledgment of the change that was submitted for review together with an URL pointing to their submission status in Gerrit (note some parallels with academic peer review). Developers also receive one or more emails from the automatic testing system, reporting the testing results of the proposed changes. All this takes place in a sequence towards the future landing of the code and its later release.

Release management constrains the code review processes. A change should be proposed a few weeks before the targeted milestone publication date in order to be reviewed in time and included in the same milestone. Furthermore, the use of release management freezes purposively minimizes the time that code reviewers spend 'drowned' in late code reviews for features proposed late. After the freezes, new feature code reviews should be rejected by the review team and postponed until the next series development opens. In order for the teams to integrate, stabilize and launch what was implemented so far, code review efforts should be limited to existing code and bug fixing submissions. Features implemented late can introduce regression bugs close to the release date, undermining the quality of OpenStack software as a whole.

6.4 Testing, gating and continuously integrating

While code reviews lead to human judgments on what code is ready to be released or not, the Quality Assurance (QA) and the Continuous Integration (CI) operations produce semi-automated judgments on whether a piece of code is of sufficient quality be released or not. The mission of Quality Assurance is to "develop, maintain, and initiate tools and plans to ensure the upstream stability and quality of OpenStack, and its release readiness at any point during the release cycle"[39]. This mission underlines the importance of automation in complex software development settings:

> "OpenStack projects have robust automated testing. In general, we believe something not tested is broken. OpenStack is an extremely complex suite of software largely designed to interact with other software that can be operated in a variety of configurations. Manual localized testing is unlikely to be sufficient in this situation."
> — OpenStack Project Team Guide, as last edited on 20 Sep 2017[40].

Developers should test their contributions properly before submitting them for review, doing not only unit or functional tests but also performing integration and performance tests against the official code base. Furthermore, style checks should also be employed for ensuring sufficient consistency on source code jointly developed by multiple developers. However, as the OpenStack code base is enormous with more than 20 million lines of code by October 2017, and as many of the OpenStack cloud computing features require vast resources (e.g., clusters of multiple machines, multiple hypervisors, and hardware-accelerated virtualization among others resources typically found in data centers), not all developers have access to sufficient resources to run all the tests. Furthermore, code reviewers cannot insure or assume that developers have performed all of the relevant testing prior to their code submissions. Many of the OpenStack

projects implement therefore batteries of tests (aka sets of testing jobs) that automatically run on every change to the project[41]. These test batteries are often developed in cooperation and coordination with the Quality Assurance, Infrastructure and Release Management teams.

In OpenStack, test automation is implemented in *Jenkins* and *Zuul*. Test results appear in http://Gerrit allowing developers to easily trace their contributions. Jenkins automation server does the most atomic work of building and executing testing jobs, including all unit, integration, and code-style tests. Jenkins is partly controlled by Zuul which determines what jobs are run and when. Zuul acts also as a gateway between Gerrit and Jenkins. In one direction, it listens to the Gerrit event stream and triggers jobs on Jenkins. In the other direction, Zuul listens to Jenkins testing jobs results and adds a machine-made review to Gerrit. Test results are reported to Gerrit by Zuul in the form of a 'machine' vote that will complement the human code-reviews. Only after the code reviews have been completed and the code passed the Jenkins/Zuul tests, the 'gate opens' and the code can finally 'land' into the official master branch. A stable branch will be cut by the end of the release cycle from this master branch.

After releasing, keeping the stable branches in good health is an ongoing effort. Developers can see what bugs are currently causing gate failures and they can prevent code from merging into stable branches. Even if the tracing of bugs varies from team to team, the Launchpad platform[42] remains the most used infrastructure for tracking bugs. As fixing bugs in stable branches often requires much cooperation and coordination among different teams, OpenStack developers rely on Etherpad[43] dynamic documents that link to information on current bugs and in-flight fixes across the OpenStack ecosystem. Developers are encouraged to discuss bug fixing issues on the official IRC and mailing lists which are archived on the OpenStack website. As pinpointed by Poo-Caamaño et al. (2017) [10], release management information should be communicated across and made available to the overall ecosystem participants.

While the code reviews orchestrated by Gerrit provide individual formative feedback on the code-contributions, the continuous integration tests run by Jenkins and Zuul provide machine-made (aka testing bots) summative feedback of the code contributions. Armisen et al. (2016) [51] studied the the interplay between formative and summative feedback at OpenStack. They suggest that summative feedback (i.e., testing bots) influences how developers take formative feedback (i.e., code reviews).

While a code review vote is highly personal and might vary from reviewer to reviewer, the automated testing results are more often unassailable. In an analogy with the academic world, we could say that code that does not pass the continuous integration tests is code that tends to be "desk rejected". While code contributions are reviewed once in Gerrit, Jenkins and Zuul run the automated tests twice, before and after the approval by the reviewers. The OpenStack community claims that continuous integrations tests are in line with the open and egalitarian nature of the OpenStack project as, after all, it should not matter where the code comes from, from which particular developer or company, it needs to pass the same tests to make it to the official code base.

> "OpenStack projects do not permit anyone to directly merge code to a source code repository. Instead, any member of a core reviewer team may approve a change for inclusion, but the actual process of merging is completely automated. After approval, a change is automatically run through tests again, and only if the change passes all of the tests, is it merged.
>
> This process ensures that the main branch of development is always working (at least as well as can be determined by its testing infrastructure). This means that a developer can, at any point, check out a copy of the repository and begin work on it without worrying about whether it actually functions.
>
> This is also an important part of the egalitarian structure of our community. With no member of the project able to override the results of the automated test system, no one is tempted to merge a change on their own authority under the perhaps mistaken impression that they know better than the test system whether a change functions correctly." — OpenStack Project Team Guide, as last edited on 20 Sep 2017[40].

OpenStack is a large, complex and very heterogeneous project aggregating software for very different purposes and written in many different programming languages. Different project teams might adopt different testing tools depending on their particular context. For example, the Horizon project that implements the OpenStack's Dashboard, which provides a web-based user interface to core OpenStack services such as Nova, Swift, and Keystone among others, adopted the Selenium user interface testing automation framework for web applications[44]. As Horizon is developed on top of the Django web framework[45] and it outputs dynamic web user interfaces, Selenium can be used to automate graphical user interface tests that run on top of the most modern web browsers. In other words, this tool allows Horizon contributors to write software 'bots' that test the evolving graphical user interface. That is, developers simulate certain user behaviors and check if the interface 'reacts' as expected.

Selenium uses the so-called locators to find and match the elements of a web page to interact with it for testing

purposes. Selenium can hook with many user interface elements of dynamic HTML pages (e.g., Id, Name, Link, DOM, XPath, CSS). Therefore the Selenium user interface testing 'bots' need to co-developed *vis-à-vis* with the user interface. Radical changes on the many elements of a graphical user interface can turn exiting user interface tests (orchestrated by Selenium in the OpenStack Horizon case) unreliable, unless the user interface and its tests are co-developed in resonance with each other. The following illustrative commit log from the OpenStack Horizon Git repository 1 illustrates the efforts of an OpenStack developer to maintain the user interface and its tests in congruence with each other. Regarding both release management and user interface testing automation, the use of the above-mentioned freezes along with the release management cycle can ease the writing of the user interface testing 'bots' as the user interface testing elements get frozen and stable.

6.5 Cross-team release management

The release management team filters, assesses, validates and releases the overall code base. The team manages the release process for the many deliverables proposed by each project team at a given development cycle. The team also provides and maintains the many tools that support the overall release management duties (see Table 2). The release management team acts as a cross-project organization that spans the different project teams. It relies on a liaison from each team project to help with coordination and release-related tasks. This release management liaison is often the Project Team Leader or someone formally designated by the Leader. Release management liaison should exhibit leadership especially within the project team, exhibit communication skills within and across teams, follow the release guidelines, keep track of the development cycle tasks, attend the cross-project meetings (e.g., the technical committee meetings that generally happen on Tuesdays) and ensure that known bugs are correctly reported and triaged. Release management liaisons need to deal with many of the interdependencies across the different OpenStack project teams and their repositories.

In this way, much of the work with release management remains largely manual, far from fully automated. The release management team develops and maintains a significant number tools and scripts to handle the release process, but much of the work remains manual, something that the OpenStack community acknowledges and intends to automate further. Since the release of the Liberty series (12th release of the project on 16 October 2015), OpenStack adopted a novel 'micro-tagging' approach together with a novel tool developed by the release management team to support the approach. This tool was named *Reno* and it can be seen as a release notes manager that encourages developers to provide more detailed and accurate release notes for each software deliverable being released.

Also Git has the ability to tag specific points in source-code history. These Git tags point out to a specific commit and thereafter do not change even when the corresponding Git branch moves on. While commits are identified

Table 2 Tools supporting release management at OpenStack

Tool	Tool description	Roles
Launchpad	Collaboration and hosting platform	Bug tracking and discussion of blueprints.
Story board	Task tracker	Task tracking across multiple teams, repositories, and branches.
Git	Distributed code repository system	Hosting, version and revision control.
GnuPG	Hybrid-encryption software	For annotating and signing Git tags.
Gerrit	Code review system	Orchestrates peer review of proposed code changes over Git.
Jenkins	Automation server	Runs jobs of continuous iteration testing.
Zuul	Pipeline oriented gating and automation	Acts as a configurable gateway between Gerrit and Jenkins.
Logstash	Pipeline and oriented analysis of jobs	Analysis of logs.
Etherpad	Collaborative online editor (real-time)	For each stable branch, it aggregates information on current bugs, failures of the continuous integration jobs, known problems/issues, and on recently closed problems/issues as well.
Reno	Release notes manager.	Manages release notes in a standardized format.
Sphinx	Documentation and specification generator.	Integrates with Reno to generate reports containing release notes for specific branches and versions.
Mailman	Software for managing electronic mail discussion and e-newsletter lists	Official channel for discussion among developers. Results from code reviews, continuous integration testing, and code-merging are delivered by software bots via e-mail.
Freenode	IRC	Textual discussions of developer across different channels (achievable material).
MediaWiki	Wiki software	Collaborative documentation.

```
Fix Selenium Tests
A recent update to XStatic-Jasmine appears to have altered the class name
that a selenium test was looking for. This patch amends the class so that
the test correctly finds the class.
Change-Id: I8d6844cdb14a84bd5498429b8545cdca8ba173b6
Closes-Bug: 1567965
(cherry picked from commit 7661db1ba438a13cf77d64913bfc88767b6bcdae)
```

Illustrative git commit log 1 - Selenium test update.

by unique hash values, tags can be identified by textual labels[46]. OpenStack contributors use Git tags as mechanisms to define the release points with descriptive names such as "v3.1.4b1", "tc:approved-release", ":ref:'tag-vulnerability:managed' " or ":'tag-stable:follows-policy'"[47]. As pointed out before by Poo-Caamaño (2016) [33] that also studied release management at OpenStack, the recent replacement of the big 'official release' by a tagging system where each sub-project is marked as 'release managed' by the technical committee addresses possible misconceptions from the distributors, integrators, and operators of OpenStack. After all, the concept of 'official release" can lead to wrong assumptions and its users can assume incorrectly that all of these 'official release' sub-projects shared the same level of quality and maturity [33, p111]. The concept of an integrated 'official release' worked better while OpenStack was smaller project, but as the ecosystem grow the definition of a final and complex modular system becomes blurry, the tagging approach is in this sense forces the users of OpenStack to not look at the new release of OpenStack but at the new coordinated releases of OpenStack.

With the Liberty release, many different OpenStack project teams were invited to use Reno for providing release notes that are directly attached to the Git source code tree. This implies that a patch can include a Reno file, or a series of them, containing textual information explaining what is the expected impact of this particular incremental change. The standardized use of release notes as orchestrated by Reno, as well as the use of Git tags that are signed by centrally managed OpenPGP keys maintained by the OpenStack Infrastructure team, pave the way for the overall standardization, access control and automation of cross-project release management processes.

There are several important sections that a release description as a Reno file can include, to become automatically associated with the release version based on the Git tags applied to the repository. With this approach, it is not necessary to track changes manually using a bug tracker or other tool, as release notes are stored next to the code being developed and encoded in a human and machine readable way. Developers can write down their release notes at any time (hopefully while the understanding of what was developed remains fresh), store them close to the artifacts being developed, and automatically deliver them to the release management team when appropriate.

7 Discussion

In this paper, we investigated release management at OpenStack while paying special attention to its processes and infrastructural tools. Our findings complement the current body of empirical knowledge addressing release management in the context of open source software [5, 10]. As release management practices are connected to other software engineering practices such as 'planning, code-reviewing, continuous integration, quality assurance, documentation and translation, our results might connect with other issues of interest in the Software Engineering and Free/libre/open source software (FLOSS) research communities.

Prior work has already inquired on OpenStack release management issues (see [16, pp 10-11] for work bringing up collaboration issues and [33, pp 80-82] for work

prelude General comments about the release. Should succinctly announce what was developed to be included in the release. Should also mention major impacts and interdependencies with the OpenStack totality.

features Details the list of new major features in the release.

issues Includes a list of known issues in the release. For example, if a new driver is experimental or known to not work in some cases, it should be mentioned here. For features that are not crucial, it might be better to release on schedule, document known issues and fix them later.

upgrade Should be provided when a patch has an UpgradeImpact tag. Used for instance when a database changes or needs a deployer modification (e.g., a migration). Also when a configuration option changes (e.g., depreciation, removal or modified default). Overall, any patch that requires action by a deployer should include this 'upgrade' section.

critical If the patch fixes a critical bug (as classified in the Launchpad), it should be reported here.

security If the patch fixes a known vulnerability, it should be reported here.

Release notes in OpenStack - Different sections of a Reno file.

pointing out communication issues). However, and to the best of our knowledge, this is the first paper paper that explicitly aims at describing how a large and complex open source software ecosystem refined its release model over time. We longitudinally followed OpenStack technology from its inception at NASA to its bootstrap as an open source project and its more mature phase characterized by a liberal time-based release strategy where multiple cycles co-exist.

When integrating with prior related literature, our results confirm the pivotal role of freezes within the release management process (cf. [13, 37]). The use of the three freeze mechanisms by the release management team (i.e., "FeatureFreeze", "SoftStringFreeze", and "HardStringFreeze") encourages developers to progressively change their production focus from new developments to integration and stabilization of that what was developed so far (see Table 1). In our case, the use of freezes forces developers that want to see their work in the next release to make three major shifts in the focus of the production: (1) from the individual component level to the overall integration as a whole, (2) from developing new features to ensuring their landing, integration and stabilization, and (3) from individual work, or collaboration within smaller teams, to coordination across the overall community. Our investigation found the use of freezes particularly helpful for the practice of user-interface testing automation (see Section 6 for a short account on the efforts of the Horizon project team to automate the testing of dynamic HTML user interfaces).

Also, in the light of prior work, the release management process of OpenStack can be considered a hybrid of feature-based and time-based release management [32, pp 23]. In addition to regular releases every six months, OpenStack also attempts to introduce new features at each regular release. At the planning stage, leaders of each project team choose a set of features for the next release. However, if these features are not stable enough to be included in the upcoming release, they will be left out by the cross-project release management team. As pointed out recently, release management constrains the evolution of the integrated whole [33, pp 4].

By investigating both the release management processes and the tools that support it at OpenStack, we found both the micro-tagging approach and the Reno release notes manager as novel and distinctive from other open source release management cases (c.f., [5, 10],). Future research could report on how the OpenStack technical committee and the release management team employ repository tags to signal that a certain component of OpenStack was release managed (an indicator of quality), that certain component can be marketed as a core component of OpenStack, or that a given project followed a suitable design or achieved a high level of diversity in the affiliation of contributors (an indicator of a healthy collaborative project). Assuming that others might be interested in Reno as a novel open source tool supporting release management processes, future research should explore whether the following characteristics of Reno could bring value to the practices of release management in particular and software engineering in general.

- Release notes are automatically associated with the release version based on the repository tags applied to the repository. It is not necessary to track changes manually outside the repository (e.g., in a bug tracker, a spreadsheet or other tool).
- Release notes are encoded within the source code repository at the side of correspondent features source code. This means that release notes can be written when the code changes within the same development environment.
- Release notes go through the same review process used for managing code and other documentation changes.
- Release notes are encoded in a standardized format. Notes are organized into logical groups based on whether they describe new features, bug fixes, known issues, or other topics of interest to deployers, users, and developers.
- Prior to delivering new features to the release management team, release notes can be automatically aggregated and documented from the source code repository with Reno. Developers only need to run a script that invokes Reno.
- Release notes can be easily located by project, release series, branch, earliest revision, and date, among other parameters. Developers can search for specific sets of release notes and sections.

Adding to prior work, and in addition to considering the overall release management process of OpenStack as a hybrid of feature-based and time-based release management, we also consider it as quite liberal. We found the OpenStack release management process to be liberal in multiple aspects:

- Liberal as the official releases dates, the milestones, and the freezes are not strict but negotiated and applied by the release management team in cooperation with the different project teams. In an analogy with trains, a given train might be scheduled to depart at a given time, nevertheless, they might come a bit before or a bit later due to a number of organizational or technical issues. Still, the train schedules remain a useful artifact for planning purposes.

- Liberal as different teams (e.g., testing, drivers, documentation, and translation) are granted with extra days, or even a few weeks, to deliver their own releases. Releasing depends on the context, nature and interdependencies of the artifacts being developed. Examples include testing how a certain hardware driver complies with new developments, developing a new library that relies on external APIs and translating recent documentation to another language.
- Liberal as feature freeze exceptions are granted by the release management team and the technical committee in exceptional cases (e.g., for landing a release critical fix). These feature freeze exceptions need to be properly discussed and documented before being granted, controlled and monitored.
- Liberal, as well, as OpenStack started more recently recommending four different release management models. Even if most of the models trail the common six-month release 'train', the OpenStack governance recognized over time the need for certain teams and individuals to manage their own release strategy independently.

Given the complexity of the OpenStack software ecosystem in general and the historical evolution of its release management processes in particular, we opted to keep our main research efforts as descriptive. Our main contribution is then a straightforward longitudinal account of the release management practices at OpenStack, an account that, to our belief, can bring issues of interests to the Software Engineering and Free/libre/open source (FLOSS) research communities. Both academics and practitioners can now assess the similarity or dissimilarity of the release management processual patterns of OpenStack with other projects - this kind of comparison can lead to lessons learned or even to improvements on the way organizations release software.

We have studied, gained understanding and described many of the release management processes and tools with OpenStack. We have given our own interpretations of the release management phenomena in the context of OpenStack, that is, in an open source software ecosystem that has so far continuously grow in size and complexity. Our most striking finding is that the evolution of release management processes at OpenStack has led to the co-existance of not one but several release management cycles. Since October 2016, the OpenStack release management formally recognize four different release management models.

We can conclude that as a software ecosystem grows in size and complexity, its developers might follow not one but several release management cycles. This conclusion calls for further research investigating why, how, and when projects should implement multi-cycle release management strategies. In the particular case of OpenStack, it remains unexplored whether the formal co-existence of four different release management models has had a positive impact on the amount or quality of the software being contributed by its developers. Future research addressing multi-cycle release management strategies might be a fruitful avenue in Software Engineering. So far, we know that smaller open source projects at early stages traditionally announce releases once new features are implemented. We also know that large and successful open source projects benefited with the implementation of time-based release schedules [5], but so far little is known about projects where multiple release models co-exist with each other as we found in our case. Future quantitative investigation unveiling causal relationships between the socio-technical characteristics of teams and artifacts (e.g., sub-projects team size, application vs. library, modularity, and coupling among many other socio-technical characteristics) and the different release management models, are, in our view, research efforts worth being explored.

At this point, we are not attempting to evaluate, appraise or compare the captured release management processes of OpenStack. Our focus was on describing the most salient release management patterns by deeply studying them. Besides leading to descriptive findings, our investigation of OpenStack opens multiple avenues for future research.

An obvious step is to move from a single case to a multipl case study design. Future research could analyze and juxtapose the processual practices of release management across multiple cases [5, 33]. Both within OpenStack, or in other projects, digital trace data generated by the upstream integration processes, the source code repositories, the code review systems, and the bug trackers could be used to triangulate the authenticity of the conceptual release management models in practice[48]. Also, it remains unknown why and how multiple release cycles co-exist in large and complex software ecosystems. Another salient issue pertains to the management of information regarding release management (e.g., should release management notes be stored at the side of the source code in the repository, or should new systems be developed for better structuring all information regarding release management).

Given the co-existence of multiple avenues for future research, and as we intend to continue our engagement with the OpenStack community, we plan to address some of the current challenges faced by the OpenStack release management team. The OpenStack community could benefit from better co-release practices with its end users and deployers, because their feedback often comes too late to shape the next immediate release cycle. Better practices

could also reduce the large cognitive load that first time, part time and occasional contributors face releasing their software, because release management adds a set of discouraging barriers for contributors that are not that familiar with the overall research management processes. Engaged empirical research taking a critical stance on these practical issues could lead to improvements in the release management process and overall organizational design of the OpenStack ecosystem. While so far we had focused on processes and tools, future research should also explore the organizational design supporting and shaping release management at OpenStack. Along these lines, future research could engage with theoretical frameworks such as the Conway's mirroring hypothesis [52], socio-technical congruence [53] or socio-materiality [54] to explore release management from an entangled organizational and technical perspective. After all, it is expected that the organizational design supporting release management is shaped by the artifacts being developed (e.g., code and documentation) and the other way around. The use of such theoretical frameworks challenges researchers to not separate organizational and technical issues from each other but build knowledge while intertwining them.

8 Conclusions

OpenStack implemented a time-based release strategy that trails a six-month cycle. Each cycle comprehends a planning stage, an implementation stage and freeze, stabilize and launch stage. In the middle of each release cycle, the community relies on three freezes (i.e., "FeatureFreeze", "SoftStringFreeze" and "HardStringFreeze") that encourage developers to change their production focus from the development of components to the overall upstream integration and stabilization of components as a whole. This change affects much the work and communication patterns of the community. The implemented release management process exhibits hybrid characteristics of both feature-based and time-based release management strategies as the process is both feature and time oriented. Moreover, the implemented release cycle is quite liberal and in this way open to changes and flexible to adaptation. In particular contexts, different project teams across the community are allowed to work around the default six months release cycle. Even when the project advocates a six month release cycle, different release cycles do co-exist across the different OpenStack sub-projects.

The implementation of a liberal time-based release strategy is a complex process that intertwines with many other software development processes. In the case of large and complex open source software ecosystems, this requires the support of a well suited organizational design as much coordination is needed. Moreover, the process constrains the evolution of the integrated core and depends heavily on many software tools that make it possible. These tools help, for example, version control, revision control, continuous upstream integration, continuous upstream testing, and configuration management. Besides its acknowledged benefits (see [36]), the implementation of a liberal time-based release strategy is a challenging cooperative task interweaving people with processes and technology.

Endnotes

[1] See https://docs.openstack.org/project-team-guide/introduction.html for a brief overview of the history of OpenStack as provided by the community.

[2] See the historical newsgroup https://news:comp.os.linux.announce where developers announced new releases of open source software for Linux with a strong emphasis on the implemented features.

[3] See https://www.kernel.org/category/releases.html and https://www.debian.org/releases/ for information on the releases of Linux (2-3 months release cycle) and Debian (with a two years release cycle).

[4] Here we add that many automated user interface testing tools and techniques depend on the stability of certain strings (see [55, 56]) as well.

[5] See http://www.openstack.org/ for the official website.

[6] See http://451research.com/report-short?entityId=82593.

[7] See https://doughellmann.com/blog/2016/03/15/ for a blog post regarding the process automation efforts by the OpenStack release management team.

[8] A compressed archiving format that is very popular within the open source community

[9] See https://nodis3.gsfc.nasa.gov/ for the NASA Online Directives Information Systems (Agency Level Directives) and https://swehb.nasa.gov/ from the Software Working Group for accessing relevant versions of this kind of procedural documentation.

[10] See https://open.nasa.gov/blog/opensource-development-at-nasa/ for more information on how NASA has improved its engagement with the open source community.

[11] See https://www.mirantis.com/blog/openstack-project-technical-lead-interview-series-4-thierry-carrez-chair-of-the-openstack-technical-committee-release-manager/

[12] See https://ttx.re/the-diablo-1-milestone.html for more information on the switch from a 3-month cycle to a 6-month coordinated release cycle.

[13] See https://www.mirantis.com/blog/openstack-project-technical-lead-interview-series-4-thierry-carrez-chair-of-the-openstack-technical-committee-release-manager/

[14] See historical information on the exact release dates at https://releases.openstack.org/.

[15] See https://www.openstack.org/blog/2011/09/openstack-announces-diablo-release/.

[16] For an exhaustive list of OpenStack repositories see http://git.openstack.org/cgit.

[17] We acknowledge that some OpenStack components are also hosted in multiple repositories (e.g., Neutron the "network connectivity as a service" component). They are, however, exceptional cases.

[18] Transcribed from video, see [1:26–2:06] https://www.openstack.org/summit/vancouver-2015/summit-videos/presentation/the-big-tent-a-look-at-the-new-openstack-projects-governance.

[19] See http://specs.openstack.org/ for intra-project and cross-project specifications.

[20] See https://wiki.openstack.org/wiki/Blueprints for more information on how OpenStack handles its blueprints (aka design plans) that track each feature implementation.

[21] For more information on the OpenStack code-review activities, see https://docs.openstack.org/infra/manual/developers.html.

[22] See https://launchpad.net/ for more information on the adopted software collaboration platform as well as https://launchpad.net/openstack for more information on how OpenStack uses it.

[23] See http://docs.openstack.org/infra/jenkins-job-builder/ for more information on continuous upstream unit testing as well as http://docs.openstack.org/infra/zuul/ and http://docs.openstack.org/developer/tempest/ for more information on continuous upstream integration testing across interrelated projects and repositories.

[24] See http://docs.openstack.org/project-team-guide/release-management.html for more information on the release cycles.

[25] See https://wiki.openstack.org/wiki/BugTriage and https://wiki.openstack.org/wiki/Bugs for more information on bugfixing activities.

[26] See http://docs.openstack.org/project-team-guide/release-management.html for the details of each release management model.

[27] Transcribed from video, see [6:34–7:00] https://www.openstack.org/summit/vancouver-2015/summit-videos/presentation/the-big-tent-a-look-at-the-new-openstack-projects-governance.

[28] Git was created by Linus Torvalds in 2005 for development of the Linux kernel and it is now used by millions of software development projects.

[29] See http://semver.org/ for the Semantic Versioning Specification (SemVer) adopted by OpenStack that distinguishes between MAJOR, MINOR and PATCH increments.

[30] Please note that in Git, branches are essentially mere commit bookmarks.

[31] YAML is a human-readable data serialization language commonly used for configuration files. See http://yaml.org/ for more information.

[32] Contextually, the *neutron* repository hosts the core networking services of OpenStack while the *neutron-fwaas* project hosts the firewall services that heavily depend on them.

[33] See https://github.com/openstack/releases for tracking the release deliverables of OpenStack.

[34] For more information on GnuPG see https://gnupg.org/.

[35] See http://nvie.com/posts/a-successful-git-branching-model/ for a branching model that inspired how the Git branching and tagging functionalities are used in OpenStack.

[36] See the guidelines laid out in sections 4.1 and 4.13 of the OpenStack Foundation Bylaws for legal details.

[37] As pointed out by Michlmayr (2007) [35] few open source software projects have a formal post-release review process.

[38] See https://www.gerritcodereview.com/ for more information on Gerrit.

[39] See http://wiki.openstack.org/wiki/QA for the mission statement of the OpenStack Quality Assurance team.

[40] See 'OpenStack Project Team Guide' wiki page, 'Testing (QA and CI)' section at https://docs.openstack.org/project-team-guide/testing.html for the original source and more information on Quality Assurance (QA) and the Continuous Integration (CI) operations at OpenStack.

[41] For a detailed description on how the contious integration testing system of OpenStack works, see http://www.joinfu.com/2014/01/understanding-the-openstack-ci-system/ – A blog post from an OpenStack contributor affiliated with Mirantis.

[42] See https://launchpad.net/ for more information on the Launchpad software collaboration platform provided by Canonical (a top contributor to OpenStack).

[43] See http://etherpad.org/ for more information on the collaborative online editor of documents provided by the Etherpad Foundation.

[44] See http://www.seleniumhq.org/ for the Selenium official website.

[45] See https://www.djangoproject.com/ for more information on the Django web framework.

[46] See https://git-scm.com/docs/git-tag more information on the Git tag mechanisms.

[47] See https://governance.openstack.org/tc/reference/tags/index.html for an overview of these tags.

[48] See [57–61] for recent discussions on validity issues regarding the collection and analysis of digital trace data.

Abbreviations

2D: Two-dimensional; 3D: Three-dimensional; ACL: Access control list; API: Application programming interface; CI: Continuous integration: EOL: End of life; FFE: Feature freeze exception; floss: Free and libre open source software; FLOSS: Free/libre/open source software; IaaS: Infrastructure as a service; IS: Information systems; ISD: Information systems development; OSS: Open source software; MSR: Mining software repositories; NASA: National Aeronautics and Space Administration; NOSA: NASA open source agreement; OSCON: Open source convention; OSI: Open source initiative; PTLs: Project team leads; QA: Quality assurance; RC: Release candidate; SDK: Software development kit; SNA: Social network analysis; TELCO: Telecommunications companies; TC: Technical Committee

Acknowledgments

We are grateful to a number of developers that facilitated our understanding of release management at OpenStack. Some of them clarified our observations and revised particular sections of this manuscript. We are very grateful to the overall OpenStack community for developing an open, transparent and inclusive cloud computing infrastructure for big-data while being friendly towards academic research.

Funding

This work was partially supported by Liikesivistysrahasto (grants 170387 and 170388) and the DiWIL project (Impact of Information Literacy in the Digital Workplace research project) funded by the Academy of Finland and Åbo Akademi.

Availability of data and materials

This research was based on naturally occurring data (i.e., data that was generated without direct intervention from the researchers) from publicly available Internet websites. Both textual and audiovisual pieces of evidence were collected from the following URLs:

- https://www.openstack.org/
- https://www.openstack.org/software/roadmap/
- https://docs.openstack.org/
- https://docs.openstack.org/infra/manual/developers.html
- https://docs.openstack.org/infra/jenkins-job-builder
- https://docs.openstack.org/infra/zuul
- https://docs.openstack.org/developer/tempest/
- https://docs.openstack.org/project-team-guide/introduction.html
- https://docs.openstack.org/project-team-guide/testing.html
- https://docs.openstack.org/project-team-guide/release-management.html
- https://docs.openstack.org/project-team-guide/stable-branches.html
- https://docs.openstack.org/releasenotes/
- https://docs.openstack.org/releasenotes/neutron/ocata.html
- https://releases.openstack.org/
- https://releases.openstack.org/reference/release_models.html
- http://git.openstack.org/cgit
- http://git.openstack.org/cgit/openstack/governance/tree/reference/house-rules.rst
- http://git.openstack.org/cgit/openstack/releases/tree/README.rst
- http://specs.openstack.org/
- https://wiki.openstack.org/wiki/Blueprints
- https://wiki.openstack.org/wiki/BugTriage
- https://wiki.openstack.org/wiki/Bugs
- https://wiki.openstack.org/wiki/QA
- https://wiki.openstack.org/wiki/Branch_Model
- https://wiki.openstack.org/wiki/DiabloReleaseSchedule
- https://wiki.openstack.org/wiki/Governance/Foundation/Bylaws
- https://governance.openstack.org/tc/reference/tags/index.html
- https://www.openstack.org/blog/2011/09/openstack-announces-diablo-release/
- https://www.openstack.org/summit/vancouver-2015/summit-videos/presentation/the-big-tent-a-look-at-the-new-openstack-projects-governance
- https://www.openstack.org/summit/tokyo-2015/videos/presentation/herding-cats-into-boxes-how-openstack-release-management-changes-with-the-big-tent
- https://launchpad.net/openstack
- https://451research.com/report-short?entityId=82593
- https://doughellmann.com/blog/2016/03/15/
- https://nodis3.gsfc.nasa.gov/
- https://swehb.nasa.gov/
- https://open.nasa.gov/blog/opensource-development-at-nasa/
- https://www.mirantis.com/blog/openstack-project-technical-lead-interview-series-4-thierry-carrez-chair-of-the-openstack-technical-committee-release-manager/
- https://ttx.re/
- https://ttx.re/the-diablo-1-milestone.html
- https://robhirschfeld.com/2012/09/28/balastic-release/
- https://github.com/openstack/releases
- http://nvie.com/posts/a-successful-git-branching-model
- http://www.joinfu.com/2014/01/understanding-the-openstack-ci-system/
- http://www.eweek.com/cloud/openstack-moves-from-integrated-release-to-big-tent-model.html
- https://fnords.wordpress.com/2011/07/01/time-based-good-for-community/

The raw data was archived and deposited at the first author research website (http://users.abo.fi/jteixeir/pub/rel-man/jisa2019) and at the Finnish Social Science Data Archive (http://www.fsd.uta.fi/en/).

Authors' contributions

JT conducted the overall data collection. HK participated in the analysis, in the overall crafting of the manuscript and in its successive revisions. Both authors read and approved the final manuscript.

Competing interests

The author declares no competing interests.

Publisher's Note

Received: 31 January 2018 Accepted: 9 November 2018
Published online: 01 April 2019

References

1. Raymond E. The cathedral and the bazaar. Knowl Technol Policy. 1999;12(3):23–49.
2. Raymond E. The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. Newton: O'Reilly Media; 2001. http://www.catb.org/esr/writings/cathedral-bazaar/.

3. Zhao L, Elbaum S. A survey on quality related activities in open source. SIGSOFT Softw Eng Notes. 2000;25(3):54–7. https://doi.org/10.1145/505863.505878.
4. Aberdour M. Achieving quality in open source software. IEEE Softw. 2007;24(1):58–64.
5. Michlmayr M, Fitzgerald B, Stol K-J. Why and how should open source projects adopt time-based releases? Softw IEEE. 2015;32(2):55–63.
6. Barqawi N, Syed K, Mathiassen L. Applying service-dominant logic to recurrent release of software: an action research study. J Bus Ind Mark. 2016;31(7):928–40. https://doi.org/10.1108/JBIM-02-2015-0030. http://dx.doi.org/10.1108/JBIM-02-2015-0030.
7. Khomh F, Adams B, Dhaliwal T, Zou Y. Understanding the impact of rapid releases on software quality. Empir Softw Eng. 2015;20(2):336–73.
8. Choudhary V, Zhang Z. Research note—patching the cloud: The impact of saas on patching strategy and the timing of software release. Inf Syst Res. 2015;26(4):845–58.
9. Wright HK, Perry DE. Release engineering practices and pitfalls. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12. New York, USA: IEEE Press; 2012. p. 1281–4. https://doi.org/10.1109/ICSE.2012.6227099. http://dl.acm.org/citation.cfm?id=2337223.2337395.
10. Poo-Caamaño G, Knauss E, Singer L, German DM. Herding cats in a foss ecosystem: a tale of communication and coordination for release management. J Internet Serv Appl. 2017;8(1):12. https://doi.org/10.1186/s13174-017-0063-2.
11. O'Reilly T. Lessons from open source software development. Commun ACM. 1999;42(4):32–7. https://doi.org/10.1145/299157.299164.
12. Spinellis D, Szyperski C. How is open source affecting software development? IEEE Softw. 2004;21(1):28.
13. Fitzgerald B. Open source software: Lessons from and for software engineering. Computer. 2011;44(10):25–30.
14. Wuhib F, Stadler R, Lindgren H. Dynamic resource allocation with management objectives—implementation for an openstack cloud. In: Network and Service Management (cnsm), 2012 8th International Conference and 2012 Workshop on Systems Virtualiztion Management (svm). IEEE; 2012. p. 309–15.
15. Corradi A, Fanelli M, Foschini L. Vm consolidation: A real case based on openstack cloud. Futur Gener Comput Syst. 2014;32:118–27.
16. Teixeira J, Robles G, González-Barahona JM. Lessons learned from applying social network analysis on an industrial Free/Libre/Open Source Software ecosystem. J Internet Serv Appl. 2015;6(1):14. https://doi.org/10.1186/s13174-015-0028-2.
17. Ge X, Liu Y, Du DH, Zhang L, Guan H, Chen J, Zhao Y, Hu X. Openanfv: Accelerating network function virtualization with a consolidated framework in openstack. ACM SIGCOMM Comp Commun Rev. 2015;44(4):353–4.
18. Malik A, Ahmed J, Qadir J, Ilyas MU. A measurement study of open source sdn layers in openstack under network perturbation. Comput Commun. 2017;Volume 102:139–49.
19. Müller D, Herbst J, Hammori M, Reichert M. International Conference on Business Process Management (BPM 2006). In: Dustdar S, Fiadeiro JL, Sheth AP, editors. Berlin, Heidelberg: Springer; 2006. p. 368–77.
20. Stark GE, Oman P, Skillicorn A, Ameele A. An examination of the effects of requirements changes on software maintenance releases. J Softw Maint. 1999;11(5):293–309.
21. Ruhe G. Product Release Planning: Methods, Tools and Applications: CRC Press; 2010. https://books.google.fi/books?id=rVvLBQAAQBAJ.
22. Jørgensen N. Putting it all in the trunk: incremental software development in the freebsd open source project. Inf Syst J. 2001;11(4):321–36.
23. Cleveland S, Ellis TJ. Orchestrating end-user perspectives in the software release process: An integrated release management framework. Adv Hum-Comp Interact. 2014;2014:13.
24. Laukkanen E, Paasivaara M, Itkonen J, Lassenius C. Comparison of release engineering practices in a large mature company and a startup. Empir Softw Eng. 2018. https://doi.org/10.1007/s10664-018-9616-7.
25. Teixeira J. On the openness of digital platforms/ecosystems. In: Proceedings of The International Symposium on Open Collaboration, OpenSym '15. New York: ACM; 2015.
26. Howison J, Crowston K. Collaboration through open superposition: A theory of the open source way. MIS Q. 2014;38(1):29–50.
27. Hall BH, MacGarvie M. The private value of software patents. Res Policy. 2010;39(7):994–1009. https://doi.org/10.1016/j.respol.2010.04.007.
28. Rossi B, Russo B, Succi G. Analysis of open source software development iterations by means of burst detection techniques. In: Open Source Ecosystems: Diverse Communities Interacting. Berlin: Springer; 2009. p. 83–93.
29. Wiggins A, Howison J, Crowston K. Heartbeat: measuring active user base and potential user interest in floss projects. In: IFIP International Conference on Open Source Systems. Springer; 2009. p. 94–104.
30. Ihara A, Fujibayashi D, Suwa H, Kula RG, Matsumoto K. Understanding When to Adopt a Library: A Case Study on ASF Projects, Open Source Systems: Towards Robust Practices. Cham: Springer; 2017, pp. 128–38. https://doi.org/10.1007/978-3-319-57735. https://doi.org/10.1007/978-3-319-57735.
31. Michlmayr M. Quality improvement in volunteer free and open source software projects – exploring the impact of release management. PhD thesis: University of Cambridge; 2007.
32. Wright HK. Release engineering processes, their faults and failures. PhD thesis: University of Texas; 2012.
33. Poo-Caamaño G. Release management in free and open source software ecosystems. PhD thesis. Canada: University of Victoria; 2016.
34. Martinez-Romo J, Robles G, Gonzalez-Barahona JM, Ortuño-Perez M. Using social network analysis techniques to study collaboration between a floss community and a company. In: Open Source Development, Communities and Quality. NYC, USA: Springer; 2008. p. 171–86.
35. Michlmayr M, Hunt F, Probert D. Release management in free software projects: Practices and problems. In: IFIP International Conference on Open Source Systems. Springer; 2007. p. 295–300.
36. Michlmayr M, Fitzgerald B. Time-based release management in free and open source (foss) projects. Int J Open Source Softw Process (IJOSSP). 2012;4(1):1–19.
37. Anand A, Bhatt N, Aggrawal D, Papic L. In: Ram M, Davim JP, editors. Software Reliability Modeling with Impact of Beta Testing on Release Decision. Cham: Springer; 2017, pp. 121–38.
38. Teixeira J, Mian S, Hytti U. Cooperation among competitors in the open source arena: The case of openstack. In: Proceedings of the International Conference on Information Systems (ICIS 2016). Atlanta: Association for Information Systems; 2016.
39. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. Empir Softw Eng. 2008;14(2):131–64.
40. Easterbrook S, Singer J, Storey M-A, Damian D. In: Shull F, Singer J, Sjøberg DIK, editors. Selecting Empirical Methods for Software Engineering Research. London: Springer; 2008, pp. 285–311.
41. Yin RK. Applications of Case Study Research. Thousand Oaks: Sage; 2011.
42. Eisenhardt KM. Building theories from case study research. Acad Manag Rev. 1989;14(4):532–50.
43. Flynn BB, Sakakibara S, Schroeder RG, Bates KA, Flynn EJ. Empirical research methods in operations management. J Oper Manag. 1990;9(2):250–84.
44. Kozinets RV. The field behind the screen: using netnography for marketing research in online communities. J Mark Res. 2002;39(1):61–72.
45. Kozinets RV. Netnography: Doing Ethnographic Research Online. London, UK: Sage Publications Limited; 2009.
46. Yin RK. Case Study Research: Design and Methods. Applied social research methods series. Thousand Oaks: Sage Publications; 1994.
47. Teixeira J, Hyrynsalmi S. How do software ecosystems co-evolve? a view from openstack and beyond. In: Proceedings of the 8th International Conference on Software Business (ICSOB 2017). New York, USA: Springer; 2017. p. 115–30.
48. Sharma S, Sugumaran V, Rajagopalan B. A framework for creating hybrid-open source software communities. Inf Syst J. 2002;12(1):7–25.
49. Narayan N, Finis J, Li Y, Delater A. Leveraging traceability between code and tasks for code review and release management. In: Proceedings of the 7th International Conference on Software Engineering Advances (ICSEA); 2012. p. 8–14.
50. Bosu A, Carver JC, Bird C, Orbeck J, Chockley C. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. IEEE Trans Softw Eng. 2017;43(1):56–75. https://doi.org/10.1109/TSE.2016.2576451.
51. Armisen A, Majchrzak A, Brunswicker S. Formative and summative feedback in solution generation: The role of community and decision support system in open source software. In: Proceedings of the

International Conference on Information Systems (ICIS 2016). Atlanta: Association for Information Systems; 2016.
52. Kwan I, Cataldo M, Damian D. Conway's law revisited: The evidence for a task-based perspective. IEEE Softw. 2012;29(1):90–3.
53. Cataldo M, Herbsleb JD, Carley KM. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ACM; 2008. p. 2–11.
54. Orlikowski WJ, Scott SV. Sociomateriality: challenging the separation of technology, work and organization. Acad Manag Annals. 2008;2(1): 433–74.
55. Mesbah A, Van Deursen A. Invariant-based automatic testing of ajax user interfaces. In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference On. IEEE; 2009. p. 210–20.
56. Artzi S, Dolby J, Jensen SH, Moller A, Tip F. A framework for automated testing of javascript web applications. In: Software Engineering (ICSE), 2011 33rd International Conference On. IEEE; 2011. p. 571–80.
57. Howison J, Wiggins A, Crowston K. Validity issues in the use of social network analysis with digital trace data. J Assoc Inf Syst. 2012;44(2):767–97.
58. Hedman J, Srinivasan N, Lindgren R. Digital traces of information systems made researchable. In: Proceedings of the International Conference on Information Systems (ICIS 2013). Atlanta: Association for Information Systems; 2013.
59. Freelon D. On the interpretation of digital trace data in communication and social computing research. J Broadcast Elec Media. 2014;58(1):59–75.
60. Ruths D, Pfeffer J. Social media for large studies of behavior. Science. 2014;346(6213):1063–4.
61. Crowston K. In: Matei SA, Jullien N, Goggins SP, editors. Levels of Trace Data for Social and Behavioural Science Research. Cham: Springer; 2017, pp. 39–49.