Release Early, Release Often and Release on Time. An Empirical Case Study of Release Management

Jose Teixeira[1,2(✉)]

[1] Åbo Akademi, Turku, Finland
jose.teixeira@abo.fi
[2] Turku Centre for Computer Science (TUCS), Turku, Finland
http://www.jteixeira.eu

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. Nevertheless, there are very few empirical studies addressing release management in open source software development. It is already known that most open source software communities adopt either feature-based or time-based release strategies. Each of these has its advantages and disadvantages that are context-specific. Recent research reported that many prominent open source software projects have moved from feature-based to time-based releases. In this longitudinal case study, we narrate how OpenStack shifted towards a liberal six-month release cycle. If prior research discussed why projects should adopt time-based releases and how they can adopt such a strategy, we discuss how OpenStack adapted its software development processes, its organizational design and its tools toward a hybrid release management strategy — a strive for balancing the benefits and drawbacks of feature-based and time-based release strategies.

Keywords: Open-Source · OSS · FLOSS · Release management · 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 was acknowledged as a very complex process that raises many issues among the producers and users of software [6–9]. Nevertheless, 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]. After all, the freedom to study socio-technical aspect of software development contrasts open source software from the proprietary model where access to the software development team is only granted to a few.

F. Balaguer et al. (Eds.): OSS 2017, IFIP AICT 496, pp. 167–181, 2017.
DOI: 10.1007/978-3-319-57735-7_16

Given such scarcity of empirical work addressing release management in the context of open source software [5,10], we address how a particularly large, complex and high-networked open source software ecosystem implemented a time-based release strategy. Taking the case of OpenStack, a fast growing cloud computing platform that is attracting great scholarly attention recently (e.g., [14–18]), we explore a 'time-based release management strategy' implementation in practice by looking at the release management process *per se* as well as to the organizational design and the tools supporting it.

2 Prior Related Work

Within the open source context, it is known that release management affects both producers of software and its users. In one side, prior research suggested that community activity increases when the scheduled release date gets closer [19]. On the user side, new releases result in spikes of downloads [20]. After all, as noted by the early work of Martin Michlmayr that focused on release management in open source software (see [21]), release management is concerned with the delivery of products to end-users. Is therefore not surprising that some recently saw release management as a process that supports value co-creation among suppliers and consumers of software (see [6]).

As pointed out by three recent doctoral dissertations addressing release management in the context of open-source software [10,21,22], most open source software communities adopt either feature-based or time-based release strategies. Many prominent open source software projects start with sporadic releases in which developers announce the newly developed features[1]. However, as many of this projects grown in size and complexity, they start adopting time-based release strategies[2]. An early empirical study that mined the repository of a project while it adopted a time-based release strategy (i.e., the Evolution e-mail client), suggested that the adoption of a time-based released boosted the development in general terms over time in comparison to feature-based release management [23]. More recent research, based on interviews with key members of seven prominent volunteer-based open source projects, point out that many of the problems associated with release-based strategies can be overcome by employing a time-based release strategy [5]. It is getting generally accepted that when a open-source software project grows in size and complexity, a time-based release strategy should be considered.

Time-based release strategies encompass meeting a schedule, an agenda, a deadline – either a strict or more liberal ones. To enforce that software is released on time, the use of freezes (such as code freezes), will set a clear deadline to the

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

[2] 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).

software development team. If open source developers have much freedom to self-manage their own software development efforts (when comparing with traditional proprietary paradigms), the use of freezes acts in the opposite way, it constrains the developers. If new features are not implemented before the next freeze, they will not be included in the next release. Consequently, when developers realize that a set of new features will bot be ready before the next freeze, the development of such features is either canceled, put on hold or developed at the side to be integrated later on future software releases.

Such freezes, that occur before the scheduled time-based release, act as control mechanisms that slowly halt the production of the development core code (see [13,24]). In large and complex open source software projects involving a modular architecture in which many components integrate with each other, such freeze forces developers to (1) fix and release the individual components upstream, (2) integrate the different components and test the integrated core.

As earlier reported (see [13]) such freeze categories can include:

feature freeze "no new functionality can be added, the focus should is 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;"[3]

code freeze "permission is required to make any change, 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 would be no alternatives, all cloud computation would run in hardware and software infrastructures controlled by very few players with increased customer lock-in (see [16]).

Competing with the providers of such services, the leading product alternatives are not commercial but rather four open source projects (i.e., 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 — multiple firms participate in its development as well as multiple firms attempt to capture value from it.

Our empirical unit of analysis, OpenStack is an open source software cloud computing infrastructure capable of handling big data. It is primarily deployed as an "Infrastructure as a Service" (IaaS) solution. It started as a joint project of Rackspace, an established IT web hosting company, and NASA, the well-known U.S. governmental agency responsible for the civilian space program, aeronautics

[3] Here we add that many automated user-interface testing tools and techniques depend on the 'steadiness' of certain strings (see [25,26]).

and aerospace research. The project attracted much attention from the industry. By the end of 2016, OpenStack counted with more than 67000 contributors, 649 supporting companies. Furthermore, more than 20 millions lines of code were contributed from 169 countries[4].

Both private companies (e.g., AT&T, AMD, Canonical, Cisco, Dell, EMC, Ericsson, HP, IBM, Intel, and NEC, among many others) and public entities (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 in an open source way (i.e., emphasizing development transparency while giving up intellectual property rights). Paradoxically, even if OpenStack emphasizes collaboration in the joint-development of a large open source ecosystem, there are many firms directly competing with each other within the community. 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 complementing OpenStack (e.g., VMware, Citrix, and Cisco) (see [16,27]).

We decided to address OpenStack due to its perceived novelty, its high 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 ($1.7 bn, by 2016[5]), its complexity (i.e., involving different programming languages, different operating systems, different hardware configurations) and size (20 millions lines of code contributed by more than 67000 developers).

From the early beginnings, and while OpenStack was growing (e.g., in terms of the number of contributors, its code-base, and adoption among other socio-technical indicators), it adopted a 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 organizational design and the tools supporting it.

Our efforts were built on top of publicly-available and naturally-occurring archival data derived from the OpenStack project. Such data are not a consequence of our own actions as researcher, but are created and maintained by the OpenStack community in their own pursuits of developing a cloud computing

[4] See http://www.openstack.org/ for the official website.
[5] See http://451research.com/report-short?entityId=82593.

infrastructure. We took into account many methodological notes in case study research that legitimate the use of archival data when studying a case [28–32].

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. The selection of the initial sources (i.e., departure points) took in consideration key guidelines on how to conduct qualitative empirical research online [33,34]. From the initial sources, we were forced to follow many links to collect further information related to release management in OpenStack — we often landed in blogs maintained by organizations and individuals that recurrently contribute to OpenStack. Relevant data was meticulously organized withing a database for later analysis [35, 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 software development processes.

After getting familiar with many social-technical issues within OpenStack, we analyzed the collected data from the lenses of extant knowledge in release management and open source software. Given the lack of empirical knowledge addressing release management in open source software [5, 10], we explored a 'time-based release management strategy' in practice. Our rich description on how OpenStack implemented its six-month, time-based release cycle with frequent development milestones should increase our ability to understand and explain release management within the context of complex open source software ecosystems. To enhances the validity of our description on how OpenStack implemented its time-based release strategy, we asked four OpenStack developers (two of them with release management responsibilities) to early read and comment our Sect. 5.1 in advance — we reduced then possible misinterpretations of the collected natural occurring data.

5 Results

Although our research is still at preliminary stage, we believe that some of our preliminary results can already contribute towards a better understanding of release management within complex open source software ecosystems. After all, release management is an under-researched area in which many lessons can be learned from open source software [13]. Our description of the implementation of a 'time-based release management strategy' in the particular case of OpenStack is organized as a complex socio-technological process and as a complex inter-organizational arrangement supported by different tools and systems.

5.1 Release Management at OpenStack

OpenStack was first launched by Rackspace and NASA in July 2010 as an "open-source cloud-software initiative". The first release, code-named 'Austin', appeared four months later, 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. On the other side, 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 periods that ranged from three to five months[6]. 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[7].

OpenStack is so far orchestrated by the Git distributed version control system (aka repository) and the Gerrit revision control system (aka code review tool). Its source-code is hosted across dozens of repositories[8]. 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 on "when and where are bugs introduced" or "tracing longitudinally the development of features". Moreover, by using a multiple-repository approach access-control could be customized to each individual repository, new developers would 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, each repository was then managed by the project team responsible by each component[9]. Some components, such as the OpenStack Compute (aka Nova and the computing fabric controller), are core components in which many other components rely on. To be able to integrate with such 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

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

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

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

[9] 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.

> 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 this projects coming in." — Sean Dague, 15 May 2015[10]

OpenStack keeps refining its release management process but always committed to a six-month release cycle. Each release cycle encompasses: planning (1 month), implementation (3 months), and integration where most pre-release critical bugs should be fixed (2 months). 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 bug-fixing mode (as reported in other open source projects [5,19,23]). 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 bug-fixing 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 focus 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[11] that should precisely describe 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 blueprint[12], 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, 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

[10] 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.

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

[12] See https://wiki.openstack.org/wiki/Blueprints for blueprints that track each features implementation.

branch, it is pushed to OpenStack's Gerrit review system for public review[13]. 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[14] 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. Even if the *big* corporate contributors are naturally more influential in the election of Project Team Leads (PTLs) steering the tracking process, it 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[15].

At the last development milestone OpenStack applies three feature freezes (i.e., *FeatureFreeze*, *SoftStringFreeze* and *HardStringFreeze* 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'[16]). Contributors are encouraged to turn most of their attention to testing the result of the development efforts and fix release-critical bugs. Critical missing features, dubious features, and bugs are documented, filed and prioritized. Contributors are advised to turn their heads to the quality of the software and its documentation. The development becomes mainly bug-fixing oriented and a set of norms and tools guide this last product-stabilization phase[17]. Between the last milestone and the publication of the first release candidate, contributors are incited 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 release-critical bugs are fixed, OpenStack produces the first release candidate for that project

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

[14] 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.

[15] 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.

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

[17] See https://wiki.openstack.org/wiki/BugTriage and https://wiki.openstack.org/wiki/Bugs for more information on bug-fixing activities.

(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 — 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'.

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 changed of 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 changes after RC1 should be discussed with the translation team

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 introduces any new release-critical fix discovered until 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 freeze exceptions (FFE). 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 it gets closer to the final release date, 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 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. But 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 freely merged again, 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 was 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 stuff.

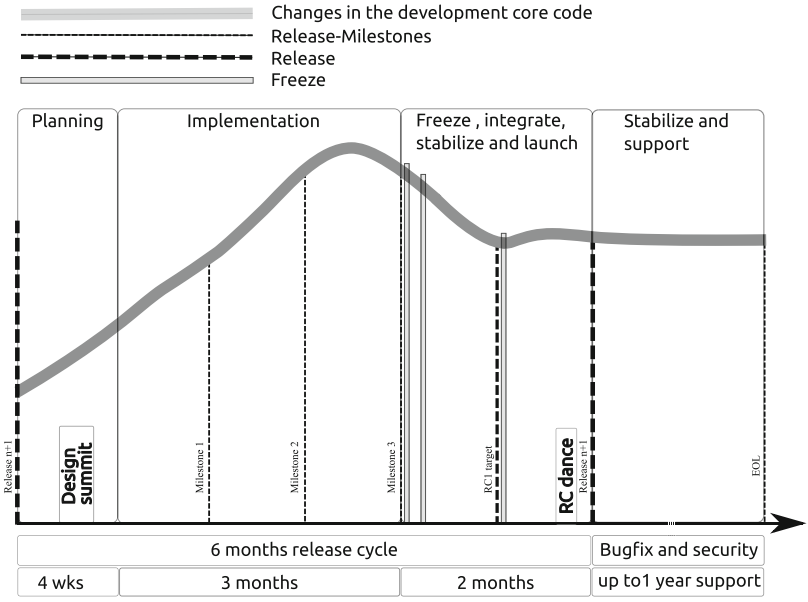

Fig. 1. Overview of the OpenStack standard release cycles.

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 the October 2016 (affecting the 'Newton' release), OpenStack actually recommends its project teams to opt from four different release management models: *Common cycle with development milestones*, *Common cycle with intermediary releases*, *Trailing the common cycle* and *Independent release model* as following described. Most of this models follow a common six-month

development cycle, some release intermediary releases within the six-months cycle and others are allowed to manage their own release strategy[18].

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 which want 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 to 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 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 instance for the OpenStack 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 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 easier. 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[19]

6 Discussion

Prior work had already inquired on OpenStack release management issues (see [16, pp. 10–11] for work pointing up collaboration issues and [10, pp. 80–82] for work pointing up communication issues). However and to the best of our

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

[19] 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.

knowledge, this is the first paper that is explicitly aimed at describing how a large and complex open-source software ecosystem implemented a liberal time-based release strategy. As this point, we are not attempting to evaluate, appraise or compare it — we are just describing it. Future research could contrast the processual practices of release management across multiple cases (see [5,10]). Digital trace data generated by the upstream integration processes, the source-code repository and the bug tracker could be used to triangulate the authenticity of the conceptual release management models.

Our results confirm the pivotal role of freezes within the release management process (cf. [13,24]). In our case, the use of freezes forces developers that want to see their work in the next release to make three big 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 its landing, integration and stabilization, and (3) from individual work, or collaboration within smaller teams, to coordination across the overall community.

Finally, in the light of prior work, the liberal release management process of OpenStack can be considered a hybrid of feature-based and time-based release management (see [22, pp. 23]). This as OpenStack encourages regular releases (every six months) but also attempts to plan the introduction of new features at each regular release. Leaders of each project team choose a set of features for the next release at the planning stage. However, if these features are not stable enough to be included in the next 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 [10, p. 4].

7 Conclusion

OpenStack implemented a time-based release strategy on a six-month release cycle. Each cycle comprehended a 'planning stage', an 'implementation stage' and 'freeze, stabilize and launch' stage. At the middle of each release cycle, the community relies upon three freezes (i.e., "FeatureFreeze", "SoftStringFreeze" and "HardStringFreezee") that encourages developers to change their production focus from the development of components to the overall upstream integration and stabilization of components as a whole — thus affecting much the work and communication patterns of the community. The implemented release cycle is quite liberal (i.e., flexible to adaptation), in particular contexts, different project teams across the community are allowed adapt the 'default' six months release cycle. Moreover, 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.

In the case of large and complex open-source software ecosystem, the implementation of a time-based release strategy, as a complex process that intertwines with many other software development processes, requires the support of a well suited organizational design as much coordination is needed. Moreover, the process constrains the evolution of integrated core and depends heavily on

many software tools that make it possible (e.g., version control, revision control, continuous upstream integration, continuous upstream testing, and configuration management). Besides its acknowledged benefits, the implementation of a time-based release strategy is a challenging cooperative task involving multiple people and technology.

References

1. Raymond, E.: The Cathedral and the Bazaar. Knowl. Technol. Policy **12**(3), 23–49 (1999)
2. Raymond, E.: The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly Media, Sebastopol (2001)
3. Zhao, L., Elbaum, S.: A survey on quality related activities in open source. SIGSOFT Softw. Eng. Notes **25**(3), 54–57 (2000)
4. Aberdour, M.: Achieving quality in open-source software. IEEE Softw. **24**(1), 58–64 (2007)
5. Michlmayr, M., Fitzgerald, B., Stol, K.J.: Why and how should open source projects adopt time-based releases? IEEE Softw. **32**(2), 55–63 (2015)
6. Barqawi, N., Syed, K., Mathiassen, L.: Applying service-dominant logic to recurrent release of software: an action research study. J. Bus. Ind. Market. **31**(7), 928–940 (2016)
7. Khomh, F., Adams, B., Dhaliwal, T., Zou, Y.: Understanding the impact of rapid releases on software quality. Empir. Softw. Eng. **20**(2), 336–373 (2015)
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. **26**(4), 845–858 (2015)
9. Wright, H.K., Perry, D.E.: Release engineering practices and pitfalls. In: 2012 34th International Conference on Software Engineering (ICSE). pp. 1281–1284, June 2012
10. Poo-Caamaño, G.: Release management in free and open source software ecosystems. Ph.D. thesis, University of Victoria, Canada (2016)
11. O'Reilly, T.: Lessons from open-source software development. Commun. ACM **42**(4), 32–37 (1999)
12. Spinellis, D., Szyperski, C.: How is open source affecting software development? IEEE Softw. **21**(1), 28 (2004)
13. Fitzgerald, B.: Open source software: lessons from and for software engineering. Computer **44**(10), 25–30 (2011)
14. Wuhib, F., Stadler, R., Lindgren, H.: Dynamic resource allocation with management objectives-implementation for an openstack cloud. In: 2012 8th International Conference on Network and Service Management (CNSM) and 2012 Workshop on Systems Virtualiztion Management (SVM), pp. 309–315. IEEE (2012)
15. Corradi, A., Fanelli, M., Foschini, L.: VM consolidation: a real case based on openstack cloud. Futur. Gener. Comput. Syst. **32**, 118–127 (2014)
16. Teixeira, J., Robles, G., González-Barahona, J.M.: Lessons learned from applying social network analysis on an industrial free/libre/open source software ecosystem. J. Internet Serv. Appl. **6**(1), 14 (2015)
17. Ge, X., Liu, Y., Du, D.H., Zhang, L., Guan, H., Chen, J., Zhao, Y., Hu, X.: OpenANFV: Accelerating network function virtualization with a consolidated framework in openstack. ACM SIGCOMM Comput. Commun. Rev. **44**(4), 353–354 (2015)

18. Malik, A., Ahmed, J., Qadir, J., Ilyas, M.U.: A measurement study of open source SDN layers in openstack under network perturbation. Comput. Commun. (2017)
19. Rossi, B., Russo, B., Succi, G.: Analysis of open source software development iterations by means of burst detection techniques. In: Boldyreff, C., Crowston, K., Lundell, B., Wasserman, A.I. (eds.) OSS 2009. IFIPAICT, vol. 299, pp. 83–93. Springer, Heidelberg (2009)
20. Wiggins, A., Howison, J., Crowston, K.: Heartbeat: measuring active user base and potential user interest in FLOSS projects. In: Boldyreff, C., Crowston, K., Lundell, B., Wasserman, A.I. (eds.) OSS 2009. IFIPAICT, vol. 299, pp. 94–104. Springer, Heidelberg (2009)
21. Michlmayr, M.: Quality improvement in volunteer free and open source software projects - exploring the impact of release management. Ph.D. thesis, University of Cambridge (2007)
22. Wright, H.K.: Release engineering processes, their faults and failures. Ph.D. thesis, University of Texas (2012)
23. Martinez-Romo, J., Robles, G., Gonzalez-Barahona, J.M., Ortuño-Perez, M.: Using social network analysis techniques to study collaboration between a floss community and a company. In: Russo, B., Damiani, E., Hissam, S., Lundell, B., Succi, G. (eds.) OSS 2008. IFIPAICT, vol. 275, pp. 171–186. Springer, Boston (2008)
24. Anand, A., Bhatt, N., Aggrawal, D., Papic, L.: Software reliability modeling with impact of beta testing on release decision. In: Ram, M., Davim, J.P. (eds.) Advances in Reliability and System Engineering. Management and Industrial Engineering, pp. 121–138. Springer, Cham (2017)
25. Mesbah, A., Van Deursen, A.: Invariant-based automatic testing of AJAX user interfaces. In: IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009, pp. 210–220. IEEE (2009)
26. Artzi, S., Dolby, J., Jensen, S.H., Moller, A., Tip, F.: A framework for automated testing of Javascript web applications. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 571–580. IEEE (2011)
27. 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). Association for Information Systems (2016)
28. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. **14**(2), 131–164 (2008)
29. Easterbrook, S., Singer, J., Storey, M.A., Damian, D.: Selecting empirical methods for software engineering research. In: Shull, F., Singer, J., Sjøberg, D.I.K. (eds.) Guide to Advanced Empirical Software Engineering, pp. 285–311. Springer, London (2008)
30. Yin, R.K.: Applications of Case Study Research. Sage, London (2011)
31. Eisenhardt, K.M.: Building theories from case study research. Acad. Manag. Rev. **14**(4), 532–550 (1989)
32. Flynn, B.B., Sakakibara, S., Schroeder, R.G., Bates, K.A., Flynn, E.J.: Empirical research methods in operations management. J. Oper. Manag. **9**(2), 250–284 (1990)
33. Kozinets, R.V.: The field behind the screen: using netnography for marketing research in online communities. J. Market. Res. **39**, 61–72 (2002)
34. Kozinets, R.V.: Netnography: Doing Ethnographic Research Online. Sage Publications Limited, London (2009)
35. Yin, R.: Case Study Research: Design and Methods. Applied Social Research Methods Series. Sage Publications, London (1994)