École doctorale 386 : Sciences mathématiques de Paris centre

Université de Paris, IRIF

Inria, Équipe-projet $\pi.r^2$

Thèse de doctorat en informatique

Challenges in the collaborative evolution of a proof language and its ecosystem

Par **Théo Zimmermann**

Dirigée par

Hugo Herbelin Directeur de recherche Inria à l'Université de Paris	Directeur de thèse
Yann Régis-Gianas Maître de conférence HDR à l'Université de Paris	Co-directeur

Présentée et soutenue publiquement le 12 décembre 2019,
devant un jury composé de :

Benoît Combemale Professeur à l'Université Toulouse-Jean Jaurès	Rapporteur
Jean-Rémy Falleri Maître de conférence HDR à l'Université de Bordeaux	Rapporteur
Pierre Courtieu Maître de conférence au Conservatoire national des arts et métiers	Examinateur
Anne Etien Maître de conférence HDR à l'Université de Lille	Examinatrice
Tom Mens Professeur à l'Université de Mons	Examinateur
Ralf Treinen Professeur à l'Université de Paris	Examinateur

Abstract

In this thesis, I present the application of software engineering methods and knowledge to the development, maintenance, and evolution of Coq —an interactive proof assistant based on type theory— and its package ecosystem. Coq has been developed at Inria since 1984, but has only more recently seen a surge in its user base, which leads to greater concerns about its maintainability, and the involvement of external contributors in the evolution of both Coq and its ecosystem of plugins and libraries.
Recent years have seen important changes in the development processes of Coq, of which I have been a witness and an actor (adoption of GitHub as a development platform, first for its pull request mechanism, then for its bug tracker, adoption of continuous integration, switch to shorter release cycles, increased involvement of external contributors in the open source development and maintenance process). The contributions of this thesis include a historical description of these changes, the refinement of existing processes, the design of new processes, the design and implementation of new tools to help the application of these processes, and the validation of these changes through rigorous empirical evaluation.
Involving external contributors is also very useful at the level of the package ecosystem. This thesis additionally contains an analysis of package distribution methods, and a focus on the problem of the long-term maintenance of single-maintainer packages.

Keywords: empirical software engineering, proof assistant, Coq, open source, open collaboration, software maintenance and evolution, GitHub, package ecosystem.

Résumé

Dans cette thèse, je présente l’application de méthodes et de connaissances en génie logiciel au développement, à la maintenance et à l’évolution de Coq —un assistant de preuve interactif basé sur la théorie des types— et de son écosystème de paquets. Coq est développé chez Inria depuis 1984, mais sa base d’utilisateurs n’a cessé de s’agrandir, ce qui suscite désormais une attention renforcée quant à sa maintenabilité et à la participation de contributeurs externes à son évolution et à celle de son écosystème de plugins et de bibliothèques.
D’importants changements ont eu lieu ces dernières années dans les processus de développement de Coq, dont j’ai été à la fois un témoin et un acteur (adoption de GitHub en tant que plate-forme de développement, tout d’abord pour son mécanisme de *pull request*, puis pour son système de tickets, adoption de l’intégration continue, passage à des cycles de sortie de nouvelles versions plus courts, implication accrue de contributeurs externes dans les processus de développement et de maintenance *open source*). Les contributions de cette thèse incluent une description historique de ces changements, le raffinement des processus existants et la conception de nouveaux processus, la conception et la mise en œuvre de nouveaux outils facilitant l’application de ces processus, et la validation de ces changements par le biais d’évaluations empiriques rigoureuses.
L’implication de contributeurs externes est également très utile au niveau de l’écosystème de paquets. Cette thèse contient en outre une analyse des méthodes de distribution de paquets et du problème spécifique de la maintenance à long terme des paquets ayant un seul responsable.

Mots-clés : génie logiciel empirique, assistant de preuve, Coq, open source, collaboration ouverte, maintenance et évolution du logiciel, GitHub, écosystème de paquets.

Extended abstract

Research on programming languages and underlying logical foundations has led to the design of programming languages with incredibly powerful type systems that give much stronger guarantees to programmers. During the same time, research on software engineering has led to technical and methodological advances that have made programmers incredibly more productive. Unfortunately, there have been too few bridges between the two domains. Some of the most advanced programming languages have been developed in academic teams with little knowledge on software engineering, and even less regarding recent research progress. In this thesis, I present the application of software engineering methods and knowledge to the development, maintenance, and evolution of Coq —an interactive proof assistant that may also be viewed as a programming language with a very strong type system (based on type theory)— and its package ecosystem. Coq has been developed at Inria since 1984, but has only more recently seen a surge in its user base, which leads to much stronger concerns about its maintainability, and the involvement of external contributors in the evolution of both Coq as a proof language, and its ecosystem of plugins and libraries.

Recent years have seen important changes in the development processes of Coq, of which I have been a witness and an actor. The contributions of this thesis include a historical description of these changes, the refinement of existing processes, and the design of new ones, the design and implementation of new tools to help the application of these processes, and the validation of these changes through empirical evaluation.

The switch from a push-based to a pull-based development model was first motivated by the opening to new contributors, but has also affected the quality of integrated changes. I present a historical overview of the adoption of pull-based development, and the changes that we implemented following this switch, including the systematic testing of external reverse dependencies through continuous integration, the use of labels, pull request templates, and the involvement of external contributors in the pull request integration process. Other changes include a switch of bug tracker, and the adoption of shorter release cycles. Each of these changes had associated challenges, which I identified and contributed to addressing. For instance, switching bug trackers required migrating preexisting issues and preserving meta-data as much as possible, which I achieved by reusing and adapting an existing migration tool. Switching to shorter release cycles required inventing a process and implementing associated tooling to manage efficiently the backporting process and the preparation of release notes. The solutions that I describe would be easy to apply beyond the Coq project.

To empirically validate some actions that were taken, I have used a technique for causality analysis, imported from econometrics, on mined software repository data. I show that the switch of bug trackers, from Bugzilla to GitHub, resulted in more activity by core developers, and more external participation in discussions on bug reports. Using the same technique, I show that the introduction of a pull request template resulted in a higher proportion of pull requests including documentation, tests, and a changelog entry. This technique is presented in much detail, and could easily be applied to many more studies based on mined software repository data, in the context of empirical software engineering, to obtain more compelling results that go beyond the identification of correlations.

Finally, involving external contributors is also very useful at the level of the package ecosystem. This thesis contains an analysis of package distribution methods, and a focus on the problem of the long-term maintenance of single-maintainer packages. I identified an emerging model of community organizations addressing this problem, which I present both abstractly and with specific examples. I founded an organization based on this model in the context of the Coq package ecosystem, and it is already quite successful.

Remerciements

Je souhaite tout d'abord remercier Jean-Rémy Falleri et Benoît Combemale d'avoir accepté de rapporter ma thèse peu de temps après m'avoir rencontré, alors que je faisais mes premiers pas dans les évènements[1] de la communauté française du génie logiciel, au deuxième semestre de ma dernière année de thèse.

Ces remerciements s'étendent naturellement aux autres membres de mon jury, Anne Etien, rencontrée à l'IRIF en mars 2019, puis à Toulouse et à Cleveland, et qui est devenue pour moi un point de repère dans la communauté, Tom Mens, dont je n'ai commencé à découvrir les travaux qu'à la toute fin 2018, mais qui est sans doute l'auteur sur lequel je m'appuie le plus dans cette thèse, Pierre Courtieu, que j'ai bien plus côtoyé sur GitHub que dans le monde physique, malgré notre proximité géographique, et Ralf Treinen, irréductible représentant de l'étude des logiciels libres et des vrais systèmes logiciels à l'IRIF.

Je suis plus généralement reconnaissant pour le très bon accueil que j'ai reçu dans la communauté du génie logiciel, quand j'ai cherché à m'y insérer tardivement, et dans celle des langages de programmation et de l'informatique fondamentale, qui m'avait porté jusque-là. Je remercie en particulier Gabriel Radane de m'avoir fait découvrir tardivement[2] l'existence du GDR GPL.

La rigueur des travaux d'évaluation empirique dans cette thèse doit énormément à Annalí Casanueva Artís. Notre amitié est ancienne, mais notre collaboration scientifique remonte seulement à septembre 2018. Moins de deux semaines avant de soumettre la première version de l'article sur le *bug tracker*, j'ai fait appel à elle[3] pour obtenir des conseils en analyse de données. Elle m'a non seulement présenté la méthode d'analyse causale que j'allais utiliser à deux reprises dans cette thèse, mais elle m'a également formé en économétrie et a grandement participé à la maturation de cet article, devenu le nôtre, jusqu'à la version qui a été finalement publiée à ICSME, et ce alors qu'elle-même doit avancer sa propre thèse sur un tout autre sujet.

L'aboutissement de cette thèse doit également beaucoup à Yann Régis-Gianas, co-directeur de la dernière année, qui a eu la gentillesse de me prendre sous son aile, et de m'aider à percer dans le nouveau domaine auquel je m'étais identifié, alors que lui-même devait écrire son HDR

[1]Journée GLE-RIMEL-LOUISE à Bordeaux en avril 2019, où j'ai rencontré Jean-Rémy, et journées nationales du GDR GPL à Toulouse en juin 2019, où j'ai rencontré Benoît ainsi que de nombreux autres membres de l'équipe DiverSE.

[2]En juin 2018, soit en fin de deuxième année!

[3]Ainsi qu'à Ambre Williams, que je remercie vivement également.

et assurer un grand nombre de responsabilités par ailleurs.

Enfin, ne croyez pas que j'oublie Hugo Herbelin, mon directeur de thèse qui m'a accueilli et accompagné dans le ~~merveilleux~~ terrifiant monde de Coq, dès 2014–2015. C'est grâce aux nombreuses heures passées avec lui, pas seulement à discuter en rebondissant l'un l'autre en d'interminables digressions, mais aussi à coder ensemble un correctif ou une addition que nous venions d'imaginer, que j'ai commencé à contribuer à Coq et à m'intéresser de près aux processus de développement. Ses propres questionnements ont nourri mon approche. Je le remercie également pour la grande liberté qu'il m'a laissé, même lorsque je lui ai annoncé, en janvier 2018, c'est-à-dire en plein milieu de ma thèse, l'orientation que j'allais donner à celle-ci, résolument empirique, et bien loin de tout ce à quoi il était habitué.

J'étends mes remerciements au reste des développeurs de Coq et son écosystème, à commencer par Maxime Dénès, avec qui j'ai collaboré sur le processus de sortie de nouvelles versions, Emilio Gallego Arias et Gaëtan Gilbert, avec qui j'ai collaboré à la mise en place et à l'amélioration de notre système d'intégration continue, Pierre Letouzey, qui m'a aidé lors de la migration du bug tracker, Guillaume Melquiond, Vincent Laporte et Pierre-Marie Pédrot, qui ont servi de cobayes au système de *backporting* que j'avais mis en place, Matthieu Sozeau dont j'ai repris et jamais fini le projet de fusion de différentes tactiques d'automatisation, et Cyprien Mangin, qui m'avait pourtant aidé à surmonter certaines difficultés qui y étaient associées, Clément Pit-Claudel et Jim Fehrle, avec qui nous avons formé l'équipe de mainteneurs de la documentation, Pierre Castéran, Karl Palmskog et Anton Trunov qui ont soutenu activement la création de coq-community, mais aussi Tej Chajed, Guillaume Claret, Jason Gross, Armaël Guéneau, Matej Košik, Erik Martin-Dorel, Pierre Roux, Michael Soegtrop, Arnaud Spiwack, Nicolas Tabareau, Enrico Tassi, etc. Remerciements particuliers à Yves Bertot, Assia Mahboubi et Cyril Cohen qui m'ont soutenu à divers moments où j'en avais besoin. Sans oublier les grands anciens, qui nous ont laissé un objet d'étude extraordinaire, et les petits nouveaux, contributeurs occasionnels aux localisations géographiques variées, trop nombreux pour être cités.

Je remercie tous les membres de l'IRIF, d'avoir créé un laboratoire chaleureux malgré la couleur douteuse des peintures et la lumière des néons. J'ai eu beaucoup de plaisir à être co-responsable du séminaire doctorant, et je tiens à remercier mon prédécesseur Clément, mes acolytes Laurent et Baptiste, et mes successeurs Chaitanya et Simon. Merci aux doctorants des différentes générations (liste nécessairement incomplète) : Cyrille, Étienne, Jovana, Ludovic, Maxime, Pierre ; Adrien, Jules, Léonard, Rémi ; Antoine, Axel, Cédric, Léo, Nicolas, Victor, Zeinab ; Alen, Farzad, Félix, Gaëtan, Hugo, Kostia, Loïc, etc.

Mon introduction à la recherche n'a pas commencé par la thèse : je veux remercier tout particulièrement Tandy Warnow, qui m'a précédemment introduit à un tout autre domaine, ainsi que les thésards de son groupe. Je remercie également un sous-ensemble strict des profs qui m'ont conduit jusqu'à la thèse : j'ai été marqué par la passion de certains enseignants, à l'école primaire, au collège, au lycée, en prépa, à l'ENS, etc.

Je remercie mes amis, connus à différentes époques : Alexandre et Bruno ; Luca et Rémi ; Édouard, Jean-Baptiste et Valentin ; Arthur, Camille et Thomas, Damir, Pierre, Pierre-Antoine, Raphaël,[4] Rémi, Séb ; Adrien, Alex, Anne-Sophie et Quentin, Antoine et Nadia, Catherine, Gaspard, Marie, Mayotte[5] et Simon, Olivier et Léopoldine, Vincent ; Ambre et Annalí, Basile, Damien, Florian, les deux Guillaume, Julie et Abel, Kenji, Ulysse et Anaëlle et tous les autres anciens du C2 ; les occupants de House of Commons, Austin, TX ; Renaud ; Max et Loriane, etc.

Je remercie ma famille, et tout particulièrement mes parents, Aline et Jean-Benoît, pour leur soutien indéfectible et crucial, ainsi que mon frère Jules, qui a su me poser les bonnes questions au bon moment et a eu une influence indéniable sur ma thèse. Enfin, je remercie Cécile pour toutes nos aventures partagées.

[4]Techniquement connu à l'époque précédente.
[5]Techniquement connue à l'époque précédente.

Table of Contents

CHAPTER 1

INTRODUCTION

1.1 Programming: humans, tools and processes

Programming is a human activity and, as such, it would be a fundamental mistake to try to understand programming without the human aspects. More generally, computer science is not just the science of computers, but of how people relate to (and around) computers as well.

Programming languages do not exist in a vacuum. They are designed for human programmers. When we say a programming language is better than another, it is generally not because it allows programmers to write more programs,[1] but because it makes it easier for them to: get started with programming; get a program running quickly; efficiently produce a program that does what it is supposed to do; or continue to reason about, understand and maintain the code of a program after many years, even when they did not author it.

But what is a programming language? It is a conceptual model, complemented by a concrete tool, the compiler or the interpreter, that allows the programmer to efficiently instruct the machine how to behave to achieve the programmer's goal. However, the compiler is not the only tool that the programmer will use for this purpose. Some other tools can turn out to be really useful or even critical to the programmer's success: editors, code formatters, build systems, debuggers, test frameworks, integrated development environments (IDE), libraries, package managers, version control systems, documentation systems, bug trackers, integrated development platforms, continuous integration systems, question and answer platforms, chat systems, and

[1]Most programming languages are Turing-complete, which means that they can express exactly the same pure programs: all the computable functions. To be completely exact, there can be differences in expressiveness between programming languages regarding what impure behaviors, i.e. what kind of interactions with the physical world, are supported. Finally, I should note that the specific programming language that I am focusing on in this thesis, Coq, happens to be a non-Turing-complete, pure, programming language.

many many more. Some of these tools may be strongly tied to the programming language,[2] and as such may play a large role in a programming language's success.

The programming language is not the only model that the programmer needs, either. When programmers work in teams, they also need a development model, which may itself include a release model. In the case of open source development, in particular, where contributions can come from anywhere, and which yields interactions between people that do not always know each other, it is all the more important that this development model is shared and well understood by all participants. In the rest of this thesis, we will use the word "process" to designate these models of expectations for human interactions around software.

1.2 Research on developing software systems: the great divide

Research on software engineering and on programming languages are two subdomains of a more general field of research related to programming. The underlying questions of this field are how to make it easier to program, how to make programmers more productive, how to ensure that software systems are correct, fast, etc.

Programming language research is focused on language design and the implementation of compilers and interpreters, while software engineering research is focused on tools and processes supporting the programming task. These aspects should be complementary, but unfortunately many programming language researchers have ignored the progress made on supporting tools, the evolution of programming development processes, and the focus given to empirical evaluation in software engineering research, while most software engineering research has stayed focused on the object-oriented programming paradigm, despite the recent trend toward the functional programming paradigm [200]. The consequence is that mainstream languages have tremendous tooling support, sometimes outweighing completely the language design issues, while novel languages developed in academia can lag far behind. New languages, like Rust [186], which take advantage of the knowledge from both worlds, have been designed mostly outside academia.

1.3 Coq: a proof language developed in academia

Coq is a proof assistant, i.e. software for interactively developing and automatically verifying mathematical proofs. The language of Coq is usually viewed in two parts. The first part is the subset called Gallina. It is a programming language with dependent types. Its type system is powerful enough to express any usual mathematical statement. Proving a mathematical statement, or equivalently finding a program of a given type, is an arbitrarily complex task which

[2]Typically, test frameworks, language-specific support within an IDE, and code formatters are tied to a particular language. Build system and package managers can be generic, or specific to a particular language (and there are indeed many application-specific package managers as we will see in Chapter 6). Continuous integration systems on the other hand are usually generic, even though several started by being targeted at a specific language (Java for Jenkins [161], Ruby for Travis [99]).

no automation could ever systematically solve. Thus Coq also provides a set of tactics, that can be used to build these proofs and programs incrementally and interactively. This second part of the language is less well-specified than Gallina, and thus poses specific compatibility challenges. Programs (or proofs) written using tactics suffer from maintenance issues [244, Section 7.2] that are reminiscent of those of programs written in dynamically typed languages. Work is in progress to provide safer tactic languages that would help address these maintenance issues [156]. In the meantime, Coq developers have to be very careful when evolving some parts of the language, to avoid breaking users' projects in intractable ways.

Coq has been developed at Inria since 1984. It has slowly opened to external users, with one of the first milestones being the release of a book about Coq in 2004 [27] (followed by many more books). The distribution of user-contributed Coq packages has started early, and has relied on a modern package manager since 2014 (cf. Section 6.5.1). Efforts have been conducted to open the development of Coq itself to external contributors anywhere in the world starting in 2015 (cf. Section 3.3, although Coq was free software and patches were accepted much earlier). Even if the development today is entirely open, transparent, distributed, and online, with the number of contributors steadily increasing and now well over a hundred, it is still the case that the main developers are Inria employees, and that Coq lacks the huge environmental support that designers of a language within a large company can benefit from. Furthermore, this is not just a matter of financial support. Given the complexity of the Coq system, a high level of expertise is needed to contribute, which excludes most software engineers without strong mathematical and computer science backgrounds, and means that, like for most scientific software [257], the developers are mainly researchers, or engineers with research experience.

As noted in the Dagstuhl manifesto "Engineering academic software" [7], while some research software projects are only episodic artifacts or prototypes meant to be associated with one or a few papers, other academic software projects become long-lived and gain a wide user base. This was the case for Coq, and it came with a transition from a co-located development model to a globally distributed one. This transition pushed the development team to address not just technical debt, but also so-called "social" [271] or "organizational" [32] debt. Coq users and developers have encountered, or are currently facing, many issues that other successful programming languages and academic software projects have encountered before, or are currently facing as well. They are related to the evolution of the language, its ease of access to newcomers (especially as Coq also tries to attract non-programmer, mathematically-inclined users), and its effectiveness for users. In particular, a tension appears regarding the evolution of the language, between its initial purpose as a research prototype welcoming experiments, and its newfound role as a stable tool on which large research or industrial projects rely.

The central thesis of this work is that successful programming languages and complex software systems will necessarily encounter what we can classify as software engineering issues, and the programming language research community would strongly benefit from incorporating early on ideas and techniques from software engineering research.

1.4 Empirical answers to practitioners' issues

My perspective as a researcher is the one of an insider within the Coq development team, who wants to help produce a better system and a thriving community, who identifies concrete issues faced by developers and users, and who uses scientific and practitioner knowledge to address them, evaluate them, and contribute back to the body of scientific knowledge. As such my work adopts the framework of insider action research [55]. The four main steps of action research are diagnosing, planning action, taking action, and evaluating action.

In line with what is advocated by Meyer [193], I start by identifying concrete issues faced by practitioners during the development, evolution and maintenance of Coq and its package ecosystem. My position as an insider allows me to discover these issues much more quickly, and contribute new research questions. I identified many more questions than I could reasonably address, and I present a number of them in this thesis. That is why a lot of chapters or sections have their dedicated "open issues and future work" (sub)sections. The conclusion additionally contains a number of more general questions, and questions that are not specific to any of the topics addressed in the chapters of this thesis.

Once an issue is clearly identified, I try to find a solution to address it, most often in collaboration with other actors (members of the development team, external contributors, or users). The process usually involves searching for existing solutions in the scientific literature, or in other open source projects, proposing an idea, gathering feedback (through public online discussion, during working groups with the development team, or through private discussion with interested individuals).

The implementation step comes after this design step. Implementation should be understood in a broad sense. It can include actual software development (within the Coq code base, or outside —for instance, the bot presented in Section 2.5), but it can also include implementing changes in processes, which is usually associated with an update or an addition to appropriate documentation, and sometimes with an announcement (e.g. announcement of the switch of bug tracker, cf. Chapter 4, or of the creation of Coq-community, cf. Section 7.5.2). Most often, this step requires having first received support for the proposed action from the development team, or the concerned people. This thesis contains a few examples where I have proposed solutions which I did not implement yet, either because I did not receive sufficiently clear support, or because they simply were not high enough on the priority list.

The last step is the validation of the solution that was implemented, through empirical evaluation. This step is important to be able to contribute back to scientific knowledge. However, empirical evaluation is often difficult, and generally takes time, so not all proposed and implemented solutions have been thoroughly evaluated. We should also note that not all solutions are worth being rigorously evaluated. Because of the difficulty associated with evaluation, solutions that are not controversial, and are already well-known in the scientific literature (e.g. the introduction of a contributing guide and internal documentation to help newcomers, cf. Section 3.4), do not justify any evaluation effort. Some changes (for instance in the management of the changelog, cf. Section 5.5) are known to solve some concrete issues because they make their occurrence impossible, but at the same time they may create other issues that are worth assessing. Sometimes, these new issues are easy to identify, and do not require a thorough investigation before carrying the next round of design and implementation of a solution. Finally, sometimes, empirical evaluation would be worthwhile and I intend to carry it out, but there has simply not been enough time since the implementation of the solution to allow it yet. For instance, the RDD methodology that I present in Section 2.3.2, and which I use to evaluate the effect of introducing a pull request template (cf. Section 3.5.3), and of switching the bug tracker (cf. Section 4.4.4), does require a sufficient time window *around* the date a change was implemented.

1.5 Academic contributions

My contributions include planning and conducting changes in the Coq development tools and processes and in the organization of the Coq package ecosystem. However, in this section, I only focus on my contributions to the scientific literature, in the domains of software maintenance and evolution, open source development and open collaboration, and empirical software engineering.

1.5.1 New research questions

I identified many concrete issues faced by practitioners. I list many of them in this thesis, but not all of them are novel research questions. Novel questions, or at least questions that were, to the best of my knowledge, virtually never studied in the scientific literature so far, include:

- How do GitHub projects manage a growing number of labels? In Section 3.5.1, I did some preliminary exploration that seems to indicate that projects manage to grow past a certain number of labels only by introducing label categories, usually denoted by shared prefixes in their names.

- How can pull request templates be instrumented by GitHub projects, and what is their impact? Sections 3.5.2 and 3.5.3 contain a first but necessarily incomplete answer to this question.

- How to test compatibility with external reverse dependencies? In Section 3.6, I highlight the importance of this question, and some specific challenges. For instance, in order to test projects that exercise recently added features, it is necessary to include projects that are actively developed. Ensuring that they can continually be used to assess compatibility breakage in incoming pull requests requires developing and testing patches when these projects are broken by intentional compatibility-breaking changes.

- How to increase the number of people involved in pull request integration? A large body of literature explores the barriers faced by newcomers and how to lower them. Some previous studies also analyze the migration path for a contributor, from placing their first contribution, to joining the core team. However, there has been less focus on encouraging regular contributors to take a more central role. This is the question treated in Section 3.7.
- What is the effect of the bug tracking environment on bug tracking activity? A large literature on bug tracking already exists, but it is rather limited when it comes to comparing bug tracking environments. Chapter 4 explores this question by measuring the impact of a switch of bug tracker.
- How to efficiently manage the process of backporting patches to release branches? Previous works have only focused on the automatic identification of candidate patches for backporting. This question is presented in Section 5.4.
- How to manage changelogs and release notes for collaborative projects beyond a certain size? Section 5.5 highlights challenges that occur for sufficiently large projects, and that impose to go beyond what standard recommendations propose.
- What are the various socio-technical options for designers of package managers and package registries? As shown in Chapter 6, previous works have focused on finding technical solutions to very specific problems, or have explored existing package ecosystems, but there is a lack of surveys on the design space that has already been explored in the very numerous package managers and registries created so far. The next step would be to be able to associate design decisions with emerging structures in the package ecosystems.
- How to alleviate the problem of the many single-maintainer packages, that can play a major role in a package ecosystem's success, but at the same time pose a risk on dependent projects because the maintainer could suddenly become unresponsive? In Section 7.2, I show that the prevalence of this type of package is high.

1.5.2 New models for addressing some of these questions

Based on collaboration with other Coq developers and users, exploration of solutions adopted in other open source projects and ecosystems, I have been able to invent or discover new models that can address some questions listed above:

- In Section 3.6.2, I present the model that we have adopted to test compatibility with external reverse dependencies. Our continuous integration tracks the development branch of many external projects, and an “overlay” system allows developers to test patches when compatibility-breaking changes are proposed. This model has increased the confidence of developers when preparing changes, because it gives a better assessment of their impact. Additionally, it has allowed the Coq development team to adopt a “no stable (ML) API” policy close to the Linux model (cf. Section 3.6.3.1).

- In Section 3.7, I present the model that we have adopted to distribute the pull request reviewing and integration process to a larger team of contributors. This model is useful to facilitate the scaling of the number of contributions.

- In Section 5.4, I present the model that I have invented (after taking into account feedback from other developers and contributors) to manage the backporting process efficiently. In Section 5.5, I present the model that I have introduced, and which was partially inspired by the GitLab project, to efficiently manage the preparation of release notes. Both of them are useful to streamline the release process, and make it easier to have a rolling release manager position.

- In Section 7.4.2, I describe the model of community organization for the long-term maintenance of packages that I have first identified in the Elm ecosystem, shown to be an emerging model in many ecosystems, and instantiated in the Coq ecosystem by creating the Coq-community organization.

1.5.3 Reusable assets

Part of the implementation of proposed solutions was quite specific to the Coq project, but some were implemented with reuse in mind, or have since then been adapted to become reusable, in particular:

- The migration tool that I have adapted to handle the migration of thousands of preexisting issues from Bugzilla to GitHub (cf. Section 4.3.2) can be reused by any project wanting to conduct a similar migration (and in fact it already has).

- The bot that I have created for the Coq project (cf. Section 2.5) is sufficiently generic to be reused in other projects. So far, other projects have relied on the instance I host to synchronize their pull requests on GitHub with a GitLab repository (and take advantage of GitLab CI). I have also been in contact with people interested in hosting their own instance of the bot for a similar usage. Additionally, the bot could be reused for projects wanting to apply the backporting process that I proposed.

Finally, all the empirical analysis pipelines that I have built are distributed as Jupyter notebooks [318–323], and the new datasets that I have fetched are made available as well. This should enable reproducing the results of this thesis and allow researchers to adapt them to explore variations of the associated questions.

1.5.4 Application of a methodology for causality analysis

In Section 2.3.2, I present in great detail a method for causality analysis on non-experimental data, called Regression Discontinuity Design, and imported from econometrics. Together with Annalí Casanueva Artís, we have applied this method to analyze the impact of the switch of Coq's bug tracker from Bugzilla to GitHub. We have shown that the switch of bug tracker provoked an increase in the bug tracking activity by main developers, and the participation of more external contributors in the bug tracking discussions. We have completed this causal, quantitative analysis, with a qualitative analysis based on interviews with Coq developers that help us interpret the results. This work is presented in Chapter 4, and was published in the 2019 International Conference on Software Maintenance and Evolution [324]. I applied this method a second time in Section 3.5.3 to evaluate the impact of introducing a pull request template with a checklist in the Coq repository. These examples show the value of this method, which I believe could be applied to many more examples in the context of empirical software engineering on mined software repository data. Using causality analysis methods can help go beyond the current norm in mining software repositories, which is the uncovering of associations between different factors, taking very large datasets into account in an attempt to compensate the fact that the results stay at the level of correlations. This type of research is extremely useful, and sometimes it is very hard to go beyond the identification of correlations. However, to be able to assert with confidence some recommendations for practitioners, we have a duty of trying to go beyond correlation analysis, as often as we can. The presentation of this method was meant to be as clear and as practice-oriented as possible, so that other researchers can use it in their own research projects.

1.6 How to read this thesis?

Beyond Chapter 2, which presents cross-cutting methods that are used in several places in this thesis, all chapters are independent. They are organized in a natural progression, but the reading order does not matter. In Chapter 3, even the four main sections can be read in any order. On the other hand, most topics are connected, and therefore, there are many cross-references. Given the wide variety of addressed topics, related work and "open issues and future work" may generally be found in relevant sections.

The thesis is divided in two main parts. Chapters 3 (adopting pull-based development), 4 (switching bug trackers), and 5 (release process) belong to the first part about the maintenance and evolution of the Coq system itself. Chapters 6 (package distribution) and 7 (package maintenance) form a second part about the organization of the Coq package ecosystem. Chapters generally comprise a historical overview more specifically focused on Coq, toward the beginning of the chapters in the first part, and toward the end in the second part.

CHAPTER 2

METHODS

2.1 Introduction

While the next chapters of this thesis are concerned with a very diverse set of research questions arising from the development, maintenance, and evolution, of Coq and its ecosystem, the methods that are used are consistent in the whole thesis, so I choose to present them here.

I describe in the following sections how software repository data was collected and analyzed, with the particular concern of showing causal impact when this could be done. I also give more technical details regarding both the analysis pipelines, and the automation that I implemented (through a bot) for the Coq project.

2.2 Origin of datasets

2.2.1 Fetching data from GitHub

Most of the software repository data is mined directly using GitHub's APIs. GitHub now provides two types of APIs. Version 3 of GitHub's API [112] is a classic REST API [94], from which information can be obtained through GET requests. Version 4 [117], that was announced in September 2016 [277], is a GraphQL API. The GraphQL query language [43] is a rich, typed language, that allows specifying exactly what information is needed. It opens the door to faster requests and a reduced bandwidth usage, compared to the REST API, because only the data that is really needed is sent. It also allows users to reduce the number of requests by batching several queries into a single one. The only limit is what the server manages to fetch before a timeout that GitHub has set to about 10 seconds. Therefore, I systematically prefer the use of the GraphQL API over the REST API. However, due to how recent it is, the GraphQL API is still missing some fields that the REST API provides. In this case only, I fall back to using the latter, but also send feedback to GitHub support on what is found missing.[1]

Here is an example of GraphQL query (from Section 7.4.4):

```
query searchOrganizations($query: String!, $cursor: String) {
  search(type: USER, query: $query, first: 50, after: $cursor) {
    userCount
    pageInfo {
      endCursor
      hasNextPage
    }
    nodes {
      ... on Organization {
        login
        name
        description
        websiteUrl
        createdAt
        membersWithRole { totalCount }
        repositories(first: 1, orderBy: {field: STARGAZERS, direction: DESC}) {
          totalCount
          nodes {
            stargazers { totalCount }
            assignableUsers { totalCount }
          }
        }
      }
    }
  }
}
```

A single request returns 50 organizations resulting of a GitHub search with a given query, and for each of them, its login, name, description, website URL, creation date, its public membership count, its repository count, and for its most starred repository, its number of stars and assignable users. Using the REST API to fetch the same information would have required a first request to the search endpoint resulting in only 30 results, and returning only the organization login (plus a number of irrelevant fields to ignore). For each resulting organization, a request to the

[1]For instance, in Section 7.4.4, I needed to fetch the creation date of GitHub organizations. I first used the REST API because the field was not available in the GraphQL schema, but I also sent feedback to GitHub staff, and they added the missing field less than a month after.

organization endpoint would have returned the name, description, website URL, creation date, and number of public repositories (plus a number of irrelevant fields). Again for each organization, a second request to the public members endpoint would have returned a list of members (limited to 30), but no total count. Thus, obtaining an exact count (if that was required) would have needed to iterate through pages of results (and sending a new request for each additional page). Getting the most starred repository would have required browsing through pages of results of the organization's repositories because the API endpoint for listing an organization's repository currently only support sorting by creation, update, or pushed date, or by name. But even assuming that sorting by number of stars was supported, getting the most starred repository would have required sending a request that would have yielded the top 30 most starred repositories, and a large quantity of data for each, including the number of stars, but not the number of assignable users. Finally, the assignee endpoint would have been used to get the number of assignable users. Unfortunately, once again, getting the number would have required going through pages of results, each of them containing various irrelevant fields about each assignable user. Overall, we can conclude that using the GraphQL API makes it possible to gather data that researchers would have never dared to request with the REST API.

I am not aware of any previous empirical software engineering study having relied on GitHub's GraphQL API. This could be due in part to the fact that this API is quite recent, and most established tools are still relying on the REST API, but also to the fact that only few software engineering papers describe their data collection in details. In an unpublished paper from 2018 [197], Mombach and Valente compare various ways of accessing GitHub's data in empirical studies: the REST API, GHTorrent, and GitHub Archive. They acknowledge the existence of the GraphQL API, and state the need to extend their comparison to this fourth method.

Every time I fetch a new dataset from GitHub, I make it publicly available, to foster reproducible research, and to make it possible to build further studies on this basis.

2.2.2 Dataset from Libraries.io

Libraries.io is a service monitoring packages from 36 different package managers and registries. They have gathered a dataset [206] that they make publicly available since June 2017. It has been used in a few empirical studies on package ecosystems already, mostly by Decan and Mens and their collaborators [57, 75–77, 311], but also very recently by Dietrich *et al.* [80]. In Section 7.2, I use this dataset to find popular packages from various ecosystems without having to interface with a diverse set of APIs.

2.3 Causality inference from mined software repository data

2.3.1 Correlation is not causation

Of course, every researcher in empirical software engineering knows that correlation does not mean causation. And yet, it is not infrequent to read papers that find an association between two variables of interest, and quickly jump to the conclusion that there is a causal relationship between them (typically the event that happened before is claimed to have had an influence on the event that happened after). The ability to predict if an event will occur based on other variables (predictors) is very useful, but only highlights the existence of a correlation between the two variables, and thus should not be used to make recommendations. The reason of the correlation might typically be a third, “hidden” variable. The use of very large datasets does not change anything to this simple fact.

For instance, if we find that the time to the first comment is a good predictor of the time to integration (pull requests that get acknowledged faster are merged faster), we cannot recommend to pull requests authors to post a comment immediately after opening their pull requests (nor to ask someone else to do it for them) in the hope that this would reduce the time it takes for their pull request to get merged. The reason the time to first comment is a good predictor is likely because of a hidden variable, such as the interest of the project maintainer for the pull request.

While no one would have seriously believed such a recommendation, and a paper making it would have certainly been promptly rejected, it is not hard to find published papers, from serious authors, that show a correlation, and then, in their discussion section or conclusion, use language that tends to imply that they have found a causation (for instance, through an abusive use of the word “effect”). The main difference with the example above is that the causation they are suggesting is generally plausible, and indeed most likely to be actually true. They just did not prove it. And this does not diminish the importance of their work, because uncovering correlations is a useful first step. Fortunately, it is sometimes possible to go beyond, as there are techniques that allow deriving causation from mined, non-experimental, (software repository) data.

The most noble way of deriving causation is through randomized experiments with a control group, and there are examples of such studies in the literature, including some going beyond laboratory experiments.[2] However, experiments are generally more costly to run than *a posteriori* analysis of mined data, and they may pose ethical problems (when either the treated group or the control group suffer from a disadvantage, and did not even know they were participating in an experiment). To derive causation from past data, researchers can resort to a number of techniques relying on natural *quasi-experiments*. Quasi-experiments mean that an observable criterion can be appropriately substituted to random assignment to create treated and non-treated groups. In their 2004 paper arguing for evidence-based software engineering [167], Kitchenham *et al.*

[2]For instance, Morgan and Halfaker [198] demonstrated the impact of the Wikipedia Teahouse on newcomer retention by creating a random control group of new editors who did not receive an invitation to join the Teahouse, and comparing the long-term retention of the two groups.

suggested that quasi-experiments should be used when actual experiments are not possible. Here, I present one of the simplest quasi-experimental techniques to understand and use: Regression Discontinuity Design (on time series). This method, which is very much used in econometrics for quantitative policy evaluation, has been little used in empirical software engineering research. It will be applied twice in this thesis, in Sections 3.5.3 and 4.4.4 (the latter behind joint work with Annalí Casanueva Artís).

2.3.2 Regression Discontinuity Design

2.3.2.1 General presentation

Regression Discontinuity Design (RDD) can be performed when we have a population of subjects, which we can differentiate on a given axis (the rating variable), and there is a discontinuity (cutoff) along this axis where the population on one side of the cutoff has received a given treatment, while the population on the other side has not. For instance, companies that are above a certain threshold of employees are suddenly subjected to a new workers protection law, or students that have performed better than a certain level on their standardized tests are accepted in a graduate program.

The main idea behind RDD is that subjects that are near the cutoff were almost as likely to fall on one side or the other, and would be very similar in the absence of treatment. Thus, the differences that can be observed between the two groups are mostly due to the treatment. The most important assumption to be able to do this kind of reasoning, and thus draw conclusions regarding the causal relation between the treatment and the observed difference, is that there are *no other discontinuities which could explain the latter in part*. See Angrist and Pischke [10, 11] for an accessible and intuitive introduction to RDD, and Lee and Lemieux [173] and Jacob *et al.* [145] for further details and a practitioner's guide to its application.

Using time as the rating variable, the cutoff is a change (treatment) that occurred past a certain date, on which the observed subjects had no control. It is easier to use because it is easy to think of such events. However, there are threats to validity that are specific to RDD on time series, as noted in a recent paper by Hausman and Rapson [130]. A first particularity is the capacity of anticipation and adaptation of subjects to the treatment, that affects directly the outcome. Time-series are also subject to potential serial auto-correlation of errors (errors at time $t+1$ are not independent from errors at time t). Failing to take such auto-correlation into account can lead to inaccurately estimate standard errors (and thus to detect a statistically significant effect where no such effect should have been observed). See Section 4.4.6.1 for counter-measures that can be applied to mitigate these specific threats to validity.

In this thesis, I give two examples (see Sections 3.5.3 and 4.4.4) of application of RDD on time series (with time as the rating variable), but I believe that RDD with other types of rating variables would also be applicable and useful in empirical software engineering research based on mined software repository data.

2.3.2.2 Detailed presentation of the application of RDD in this thesis

The evolution of the outcome variable around the cutoff is modeled by two different functions, one before and one after the switch. Their purpose is to accurately estimate the value at the cutoff in the absence and presence of treatment. Since only their value around the cutoff is relevant, we can choose to estimate them on a restricted bandwidth (time window) around the cutoff, or on a larger dataset. We also have the choice of different functional forms for these models.

In the context of RDD, the choice of the bandwidth of the analysis and the functional form of the regression is always difficult. On the one hand, choosing a larger bandwidth will increase precision because including more data points will reduce standard errors. On the other hand, including data points that are far from the cutoff can bias estimation near the cutoff if the functional form is not accurately specified, because those data points will have as much influence on the estimated regression coefficients as the points that are near the cutoff.

In this thesis, we systematically adopt a conservative approach for our main specification, with a relatively small bandwidth around the cutoff to minimize possible bias, and a simple linear model to avoid overfitting. However, as a robustness check, we also estimate a linear and a quadratic model on a larger time frame. Following Gelman et al. [109], we include only a quadratic polynomial and not a polynomial of a higher order.

This conservative approach reduces the statistical power of our analysis, increasing the probability of observing false negatives (being unable to detect an effect where such an effect exists), but increases confidence in the results.

When performing an RDD with linear functional forms, we estimate the following regression:

$$\textit{Outcome variable}_i = \gamma_0 + \gamma_1 \times \textit{Relative date}_i + \gamma_2 \times \textit{After cutoff}_i + \gamma_3 \times \textit{Relative date}_i \times \textit{After cutoff}_i + \epsilon_i$$

where $\textit{Outcome variable}_i$ is the outcome variable for observation i; $\textit{Relative date}_i$ is the number of days (or another time unit) from the date of the cutoff (zero is the first period after the cutoff); $\textit{After cutoff}_i$ is a binary variable equal to one if observation i is after the cutoff and zero otherwise; $\textit{Relative date}_i \times \textit{After cutoff}_i$ is called the interaction term; and ϵ_i is the residual error. This is equivalent to estimating the following two regressions, respectively before and after the cutoff:

$$\begin{aligned}\textit{Outcome variable}_i &= \gamma_0 + \gamma_1 \times \textit{Relative date}_i + \epsilon_i && \text{(for } i \text{ such that } \textit{Relative date}_i < 0\text{)}\\ \textit{Outcome variable}_i &= (\gamma_0 + \gamma_2) + (\gamma_1 + \gamma_3) \times \textit{Relative date}_i + \epsilon_i && \text{(for } i \text{ such that } \textit{Relative date}_i \geq 0\text{)}\end{aligned}$$

We estimate this regression by Ordinary Least Squares [305], and compute heteroscedasticity robust standard errors.[3] Coefficient γ_0 is the estimated value just before the cutoff and γ_1 the slope before the cutoff. Coefficients γ_2 and γ_3 are the estimates of interest and will tell us the jump in the outcome variable just after the cutoff and the change in slope due to the cutoff respectively. We interpret results as statistically significant if the p-value is below 0.05.

[3]The errors ϵ_i are said to be heteroscedastic if their variances differ. The standard calculations for estimation errors assume equal variances [305].

2.3.2.3 Related work

Only a couple of studies that we know of have used RDD in the context of analyzing mined software repository data [282, 313]. In both cases, the authors used it to analyze the impact of a change on hundreds of thousands of GitHub repositories (with time as the rating variable). This way of applying the method is quite unusual in the sense that the date of the event is specific to each project, and it is difficult to guarantee that there are no other discontinuities around the time of the cutoff. As a matter of fact, in the first of the two studies [313], the authors acknowledge that the introduction of continuous integration is often concomitant with other structural changes to the project, and in the second one [282], the authors carefully state that they have uncovered correlation and not necessarily causation because their observation could be the result of an underlying phenomenon (impacting both the quality and the decision of the maintainer to introduce a badge). In this thesis, I show that this method can be adequately applied to demonstrate causal impact on a smaller dataset, coming from a single software project, where it is easy to manually inspect the project's timeline and rule out the presence of other relevant discontinuities.

2.3.3 Future work

As researchers in econometry are quite experienced with dealing with natural quasi-experiments, we intend, with my co-author Annalí Casanueva Artís, to import some of their other techniques, and apply them to answer software engineering questions.

2.4 Software used to retrieve and analyze data

As recommended in the Dagstuhl manifesto “Engineering academic software” [7], let me properly credit software that this thesis relies on. This section starts with software that was used in the data analysis pipelines. The next section follows up with software that was used to implement automation supporting the development of Coq.

2.4.1 Jupyter notebooks, and reproducible research

I choose to systematically conduct and present the data fetching and analysis code in Jupyter notebooks [318–323]. Jupyter [168] is a software to produce computational documents, i.e. HTML documents interleaving formatted text, code, and code outputs. The use of computational documents strongly helps the goal of reproducible research, as it makes it easier for anyone, including reviewers but also the original researchers, to reproduce the full series of steps that were used to produce the research outputs. When some steps are outside the computational document itself, they must be documented with extra care. Computational documents also facilitate the presentation of the evidence, and as such they are considered as supplementary materials to this thesis.

To make the analysis pipelines entirely reproducible, readers must know the exact versions of all the software that was used, and possibly even the broader environment. That is why libraries are loaded and their version numbers are printed systematically in the first cell of each Jupyter notebook. Furthermore, these notebooks come with a pinned Nix [83] file, that makes the environment entirely reproducible. This type of dependency specification is supported in particular by the Binder platform [153], and thus, the Jupyter notebooks can be easily run by anyone, even without Nix installed on their systems.

2.4.2 Choice of programming language, software libraries

I chose to use Python [287] (version 3.7.4) for the data analyses, because of its rich scientific computing ecosystem [224], and its great integration in Jupyter. The other candidates that were considered were OCaml [176], because this is the language my advisors and I are most familiar with, and Stata [266], because this is the language that my co-author Annalí Casanueva Artís is most familiar with. However, OCaml's scientific computing libraries are less advanced. As for Stata, while it includes very advanced packages that allow conducting standard statistical analyses much quicker than in Python, it is a proprietary software, and relying on it would have partly defeated the objective of reproducible research.

The external Python libraries that are used are:

- Requests [241] (version 2.22.0) to fetch data from GitHub;
- Numpy [213] (version 1.17.0), Pandas [188] (version 0.25.0), Scipy [151] (version 1.3.0), and Statsmodels [225] (version 0.10.1) for the data analysis;
- Matplotlib [142] (version 3.1.1) to display the resulting graphs.

2.5 Building automation supporting developers

While I have contributed to build many development tools, including various shell scripts, my main contribution in this area is the creation of a bot to automate various tasks. This bot is free and open source software, available at `https://github.com/coq/bot` under an MIT license.

Bots, or autonomous software agents, are software programs that can act on behalf of humans to conduct what would be otherwise straightforward and repetitive tasks. They are persistent: they continuously monitor their environment, and activate themselves autonomously [97]. Bots have recently been used as aids in more and more software development teams [268].

The bot that I have developed for the Coq project now comprises many features, which are documented in the README, some of which will also be referred to later in this thesis. The first and most used feature of this bot is a synchronization between GitHub pull requests and GitLab branches to allow the project to use the continuous integration features from GitLab [118] while keeping GitHub as the central development platform. This feature is currently used by several repositories maintained by the Coq development team (the central Coq repository, and the Coq package index), by external Coq libraries (the Mathematical Components library), and at least one project outside of the Coq ecosystem. The other features are used specifically by the Coq project, although some of them could also apply to other interested projects (in particular the backport management automation discussed in Section 5.4).

2.5.1 Choice of programming language, technology, software libraries

I decided to use OCaml [176] to program this bot because it is a language that provides high safety guarantees, with its strong static type system, without verbosity, thanks to type inference. In practice, it means that we get almost no runtime failures once the bot is deployed. It is also the language that is used to develop Coq, so all Coq developers are familiar with it (and some of them are experts).

OCaml nowadays comes with a rich tooling ecosystem. I use most notably Dune [81] as the build system, the OCamlFormat tool [92] for automatic formatting (as a way of experimenting with it before deciding whether to recommend its adoption in the Coq project), Merlin [37] to provide IDE-like features in Emacs [265], and Nix [83] to specify the dependencies.

OCaml's libraries ecosystem is also well adapted to write HTTP servers and clients (and a bot is not much more than a combination of both of these). I use most notably the Cohttp library [182], combined with the Lwt library [292] for asynchronous processing of HTTP requests, the Yojson library [146] to encode and decode JSON [64], and JaneStreet's Base library [147] as a standard library replacement.

The versions of OCaml and the tools and libraries that are used are not specified here because they can be regularly updated to the latest versions to benefit from the latest features. There are no compatibility constraints regarding the build environment at the current time.

In most standard programming languages, we can find a library for interfacing with GitHub, and OCaml is no exception [112]. However, these libraries are usually nothing more than a one-to-one mapping to GitHub's REST API, whereas I prefer to use GitHub's GraphQL API [117] as often as I can to reduce the number of requests that the bot needs, and to make them more specific (these advantages of the GraphQL API were already presented in Section 2.2.1). Given that this API is very recent and mutations (i.e., requests to modify the state of a GitHub repository) were mostly not available when I started this bot, it also uses the REST API when needed, but I intend to have it entirely migrated to GraphQL eventually. The bot is not just a GitHub bot, but also a GitLab bot. GitLab has also decided to develop a new GraphQL API [119], but as of today it is only a stub. When it starts supporting the type of queries and mutations that the bot requires, I intend to migrate it to use the GraphQL API for GitLab requests as well.

Given that GraphQL is a typed query language, we can ensure at compilation time that the request is well-formed. For this purpose, I use the graphql_ppx_re library [128].

Finally, I use the cloud platform Heroku [179] to deploy the bot.

2.5.2 Architecture

The bot has grown according to the needs for automation in the Coq project, initially as a single file, and is now incrementally being rearchitectured around the idea of providing a library of base bot components that can be used in a trigger-action programming model.

Trigger-action programming [141] is a programming model that has mostly been studied in the context of smart-home automation, with the idea of providing a programming framework and mental model that is accessible to anyone. The most popular trigger-action programming platforms as of today are IFTTT and Zapier [237]. Interestingly, both of them provide a GitHub integration, and Zapier provides a GitLab integration as well, but they do not include sufficiently advanced triggers nor actions to perform the kind of things that this bot does.

Its components are of three types. Each of these types are given two names, one following the terminology of Huang and Cakmak [141], and the other one following the GraphQL terminology of the corresponding type of requests [91, Section 3.2]:

Event triggers / Subscriptions are the events that the bot listens to. When such events happen, it is GitHub / GitLab that sends requests with a payload at a specific endpoint provided by the bot (viewed as an HTTP server).

State triggers / Queries are the conditions that must be met, and the additional information that must be gathered, to perform an action. The current state is queried by the bot through one or several HTTP requests (the bot is acting as an HTTP client).

Actions / Mutations are the state-changing requests that are sent in response to some event, and some conditions being met. Again, the bot is acting as an HTTP client, and an agent on the service (GitHub or GitLab, possibly both) that is being updated.

Components of a given type for a given service are defined in a specific module. The business logic combining these various components in relevant workflows belongs to a central "callback" function (executed for each event trigger).

2.5.3 Related work

Last year, GitHub announced a new feature (currently in beta) called GitHub Actions [70]. This is a new example of the trigger-action programming model, although it has not yet been studied by the trigger-action programming research community. The building blocks can be written in any language (typically scripting languages), and run in Docker containers. The blocks are combined in a custom domain specific language based on the YAML syntax [25]. GitHub additionally provides a graphical user interface to configure the workflow without manually writing the corresponding YAML. Furthermore, they encourage sharing and reusing actions (the building blocks) between GitHub users. This is a possible alternative to the bot for all the tasks that are triggered by events internal to GitHub. On the other hand, the bot also supports triggers on other services like GitLab, which would be out of scope for GitHub Actions.

In the academic literature, research on developer support automation through bots is still relatively new, but it is gaining momentum, with the creation this year of an International Workshop on Bots in Software Engineering, and a repository listing bot-related research [3].

2.5.4 Open issues, future work

Currently the bot is run through a user account (coqbot) that was created by a Coq developer in 2015, and used for various things since then, including repository mirroring, previous attempts at manually reporting status checks, and by myself for the migration of preexisting issues to GitHub (see Chapter 4). Nowadays, GitHub's recommendation is that bots are created through GitHub Apps [116]. To encourage this migration, GitHub gives access to special resources exclusively to Apps [189]. Another advantage of Apps is that it can be easier to install for new users of the bot (no webhooks to set up manually, at least on the GitHub side), and that it gives access to webhook logs to the person in charge of deploying the app (instead of the various projects that use it, but do not maintain it).

Consequently, it seems that the way forward will be to migrate to a GitHub App. However, this may also have some drawbacks. In particular, GitHub Apps have to declare a set of permissions that they require to operate, and as the needs of automation increase in the Coq project, the App will need more and more permissions. But if other projects use the same App, they may be bothered with a prompt to accept new permissions even though these new permissions may be of no help to them. A solution could be to manage multiple Apps, requiring a subset of permissions, for each type of feature, but if they are managed by the same program, this can lead to complexity in deciding with which token to authenticate, and if they are managed by different programs, this is more work for whoever deploys and maintains them.

2.6 Conclusion

In this chapter, I have presented the methods I use throughout the thesis. This includes fetching new datasets by querying GitHub's GraphQL API, whose importance I have highlighted on an example, deriving causal effects from mined software repository through the use of RDD, a technique imported from econometrics, implementing analysis pipelines in Jupyter notebooks to enable reproducible research, and building automation to support developers through the creation of a bot. This chapter complements the description of my approach as an insider within a development team, which I have given in Section 1.4.

I hope that my presentation of GitHub's GraphQL API and of RDD will enable other researchers to take advantage of them in future research. That is why I have presented them with much detail. The bot that I have developed is also designed to be useful to practitioners, beyond the sole Coq development team.

Part I

Supporting the maintenance and evolution of Coq

CHAPTER 3

From a push-based to an open, pull-based development model

3.1 Introduction

Ever since they were invented, version control systems (VCS) have been invaluable tools for effective software development and maintenance [262].

However, for distributed collaboration around source code, and more specifically for open source development, the first generation of VCS (CVCS, i.e. centralized version control systems) were not really effective because only developers with commit-access could benefit from them [72, 249]. So the introduction of distributed version control systems (DVCS), such as git [279], has been a great step forward for collaborative and open source development. They alleviate having to worry about who has commit-access [278], and make it possible to review changes, including large changes, *before* they are pushed to the main development branch.[1]

Finally, social coding platforms (such as GitHub [296]) have simplified the submission and code review process. The general consolidation of open source development around GitHub (and competitors such as GitLab [118]) has helped to create a unified contribution workflow [87], which can benefit smaller projects without formalized contribution guidelines, and contributors, who know approximately what to expect when approaching a new project.

Empirical software engineering research has also benefited from the sudden availability of very large datasets of public and consistently-organized software repositories. This has allowed researchers to study various questions associated with the pull-based development model [62].

[1]The practice of code review is still possible, in any VCS, *after* the changes are pushed [95, Chapter 2], but conducting systematic *a posteriori* code reviews requires a much stronger culture. And some CVCS, such as SVN [14] have sufficiently good branching support to allow developer teams to adopt *a priori* code reviews.

Pull-based development is a development model based on a DVCS where anyone can clone a repository, make changes, push to a branch on their own fork (public development copy) of the project, then request the project owners to integrate (pull) their changes to the official repository. It is defined by opposition to the traditional *push-based* development model, where developers with commit rights push their changes to a central repository, and contributors without commit rights must resort to formatted patches submitted via e-mail or a bug tracker.

Social coding is an approach to software development that puts the focus on collaboration, in particular through online discussion around code, and across projects (such as when developers from related projects investigate a bug together [180]). Social coding is grounded in pull-based development on coding platforms, like GitHub, where specific review tools make it possible to discuss around code, and where it is easy to cross-reference issues and pull requests across different projects. The "social coding" motto was part of GitHub's logo from 2009 to 2011.[2]

In this chapter, I present a historical overview of the use of VCS, and the move from a push-based to a pull-based development model in the Coq project, then I present four challenges associated with the new model.

The first challenge is how to onboard new contributors. This was the initial motivation for using pull requests in the Coq project, but this alone is not sufficient to attract new contributors [79], and I explain the steps we have taken to make it easier for newcomers to join the project. The literature on newcomer onboarding is extensive [267], and most of these steps are straightforward applications of standard recommendations, so this is more of an experience report than a novel contribution.

The second challenge is to improve quality through the use of pull requests. This was the second motivation for the definitive switch to a pull-based model. I explain how we introduced labels to help reviewers, and how I organized them in categories, which allowed to create even more of them. I show that this phenomenon is not unique to the Coq project, and would deserve further investigations. Then, I present the use of pull request templates, and show how this caused an increase in the quality of pull requests. To my knowledge, this is the first analysis to show the impact of the introduction of pull request templates.

The third challenge is how to use continuous integration (CI) to test for compatibility with external reverse dependencies. I present the innovative model that we introduced, and how it

[2]From the beginning of the archival of GitHub.com by the Wayback Machine [155] in May 2008 (`http://web.archive.org/web/20080514210148/http://github.com/`), to January 2009 (`http://web.archive.org/web/20090115203245/http://github.com/`), the motto that appeared in the logo was "social code hosting". It was simplified down to "social coding" in February 2009 (`http://web.archive.org/web/20090201192503/http://github.com/`), and the logo was kept unchanged until December 2011 (`http://web.archive.org/web/20111214080929/https://github.com/`), when a new version was introduced (`http://web.archive.org/web/20111220204645/https://github.com/`).

helped developers better apply our compatibility policy, and assess which breaking changes are acceptable.

The fourth and last challenge is how to distribute the review workload. I explain how we managed to switch from a model with a single integrator, to a team of 25 integrators (larger than the “core team” which comprises 10 developers), in part by relying on the recently introduced `CODEOWNERS` GitHub feature [217]. While the Coq project is not the first one to use this feature, this is, to my knowledge, the first academic report about it.

3.2 Related work

3.2.1 GitHub studies

Since GitHub took off as the central platform for collaborative, pull-based development, many academic papers have been published studying various aspects of the practice of development on GitHub.

In the first paper to study social coding, Dabbish *et al.* [69] interviewed GitHub users and observed their use of the website. They determined that transparency in GitHub and visibility of others’ actions allowed increased collaboration and learning, for instance between project owners and their users.

Gousios *et al.* studied pull-based development from the integrator’s [123], and the contributor’s [122] perspective. They found that most projects use pull requests to sollicit external contributions, but that many also use it for reviewing all code changes, and some require multiple reviews before integrating a change. Quality is the top priority during the review process, and it is evaluated through perception and tools (but few tools are used beyond continuous integration). Integrators face challenges such as maintaining quality, especially when pull requests are too large to review properly, and rejecting contributions without alienating contributors. The challenges faced by the contributors are the unresponsiveness of maintainers, the lack of documentation or guidelines to understand the code base, of proper testing infrastructure, of a project’s roadmap, and of a list of issues labeled by difficulty level.

In 2017, Cosentino *et al.* produced a survey of the literature on GitHub [62]. They identified 80 relevant papers (most from 2013–2016, five from 2012, and one from 2010). About 70% exclusively rely on mining software repositories, while the rest use survey and interviews, possibly in addition to mining software repositories. The works that they cite found among other things that very few GitHub projects attract most code contributions and most issues, that very few developers are responsible for most code contributions, and that drive-by contributions are very common. Numerous studies (15 papers) analyzed the socio-technical predictors associated with pull request acceptance / rejection, and the time it takes to evaluate a pull request. Among the uncovered predictors, it was observed that larger pull requests take longer to review, and are more likely to get rejected, and that pull requests sent to large and mature projects are also more likely to get rejected.

3.2.2 Switch to git and GitHub

While a great number of papers studied the characteristics of GitHub's development model, very few studied the impact of switching to this platform for an older project having been previously developed in a push-based forge, without as many social features. Dias *et al.* did such kind of study on six major open source projects [79]. They observed that a switch to GitHub is not always immediately followed by an increased number of contributions or contributors, and that other factors, like having clear contribution guidelines, and being ready to review pull requests, are equally important. Roveda *et al.* [250] evaluated if the code quality of projects migrating to GitHub is affected, and did not detect any such effect.

Other researchers studied the switch of projects to DVCS. De Alwis and Sillito [72] surveyed developers to understand the perceived impact of moving from centralized to decentralized VCS: the move is decided because of anticipated benefits (among them the openness to external contributors), but the switch incurs challenges in the migration process (in particular, developers are very concerned to preserve the full history of the project) and some features are lost in the new tool (in particular, the chronologically ordered revision numbers). Muşlu *et al.* [201] performed a similar study based on interviews and a survey. They identified expectations and barriers and provided recommendations to developers wanting to conduct such a migration. Brindescu *et al.* [41] analyzed the impact of using decentralized or centralized VCS through data mining and a survey. They found most notably that the type of VCS is associated with commits of different sizes.

3.3 Historical overview: towards pull requests

From at least 1991, Coq was developed using version control systems.[3] In 1999, the repository that is still in use today (after a conversion to git [279]) was started using SVN [14], by copying and refactoring code originating from the previous CVS [66] archive. This repository now comprises about 30,000 commits.

A mirror of the SVN repository was created in February 2011 at `https://github.com/coq/coq`. This repository received a first pull request a few months later,[4] that was acknowledged with some delay with the message "This is just a mirror [...]. None of the primary developers of coq watch this repo. If you want to get this merged, you shuold [*sic*] post it to http://coq.inria.fr/bugs/". The second pull request was received a year later,[5] acknowledged by a Coq developer, and merged as an SVN revision with a delay of almost one year.

[3]Source: `https://github.com/coq/coq/blob/5f7c88d/dev/doc/archive/versions-history.tex#L111`.
[4]`https://github.com/coq/coq/pull/1`
[5]`https://github.com/coq/coq/pull/2`

The development repository was switched to git in November 2013 [177], but GitHub remained just a mirror, and the development style was still push-based, with numerous developers having got commit-access over the years.[6] From then on, pull requests were one of the officially listed ways of proposing patches (formatted patches submitted via the bug tracker were also still accepted), and the activity on GitHub was notified on the Coq developers' mailing list. In the following months, dozens of pull requests were opened by just one contributor (and a few more by three other contributors), most of them getting integrated without any comment, a few already giving rise to reviews or comments by other external contributors.[7] In September 2014, a pull request was used by a core developer for the first time in order to get a review (proofreading a documentation submission).

The core developers started to seriously advertise GitHub pull requests as a way to integrate external contributions during the first Coq Implementors Workshop in 2015.[8] A year later, during the 8.6 release process, they started regularly using pull requests to submit their own changes. Finally, after the adoption of CI (see Section 3.6) developers stopped pushing changes without going through a pull request. More than 3,000 pull requests have been merged since the beginning.

Figure 3.1 shows the growth of the number of contributors who opened pull requests on the GitHub repository. We can notice that a few core contributors opened many pull requests, and many other contributors opened a few pull requests. Out of ten contributors who opened more than 100 pull requests, five were core developers in the previous, push-based model. The other five gained push-access when the merging process was distributed (see Section 3.7).

However, this figure does not really allow the viewer to distinguish between drive-by contributors [226] and regular, but infrequent contributors. Figure 3.2 shows all the pull requests that have been opened by each contributor. On this figure, we can more clearly distinguish between the contributors who open just one pull request (about 40% of all the contributors), the ones who open a few pull requests and then disappear, and the ones who stay but contribute infrequently.

These figures were created using the data from the analysis in Section 3.5.3 (cf. the corresponding notebook [322]). The list of developers who had push-access on the repository before the switch to a pull-based development model was extracted from the previous project page, mentioned in Footnote 6, and manually matched to GitHub accounts. Two developers who gained push-access after having first contributed to Coq through pull requests are not marked as having had push-access before. Conversely, the developer who opened the very first pull request (that was completely ignored) gained push-access shortly after. Therefore, this first pull request has

[6] The page at `https://gforge.inria.fr/projects/coq` lists 54 project members, one of them (G Serpyc) being a bot account.

[7] See `https://github.com/coq/coq/pull/31` and `https://github.com/coq/coq/pull/33`.

[8] This first workshop was actually called "Coq coding sprint", and its purpose was to onboard external contributors by having them meet core developers that could answer their questions. It has been held every year since then, for three years under the name "Coq Implementors Workshop", and in 2019 under the name "Coq Users and Developers Workshop".

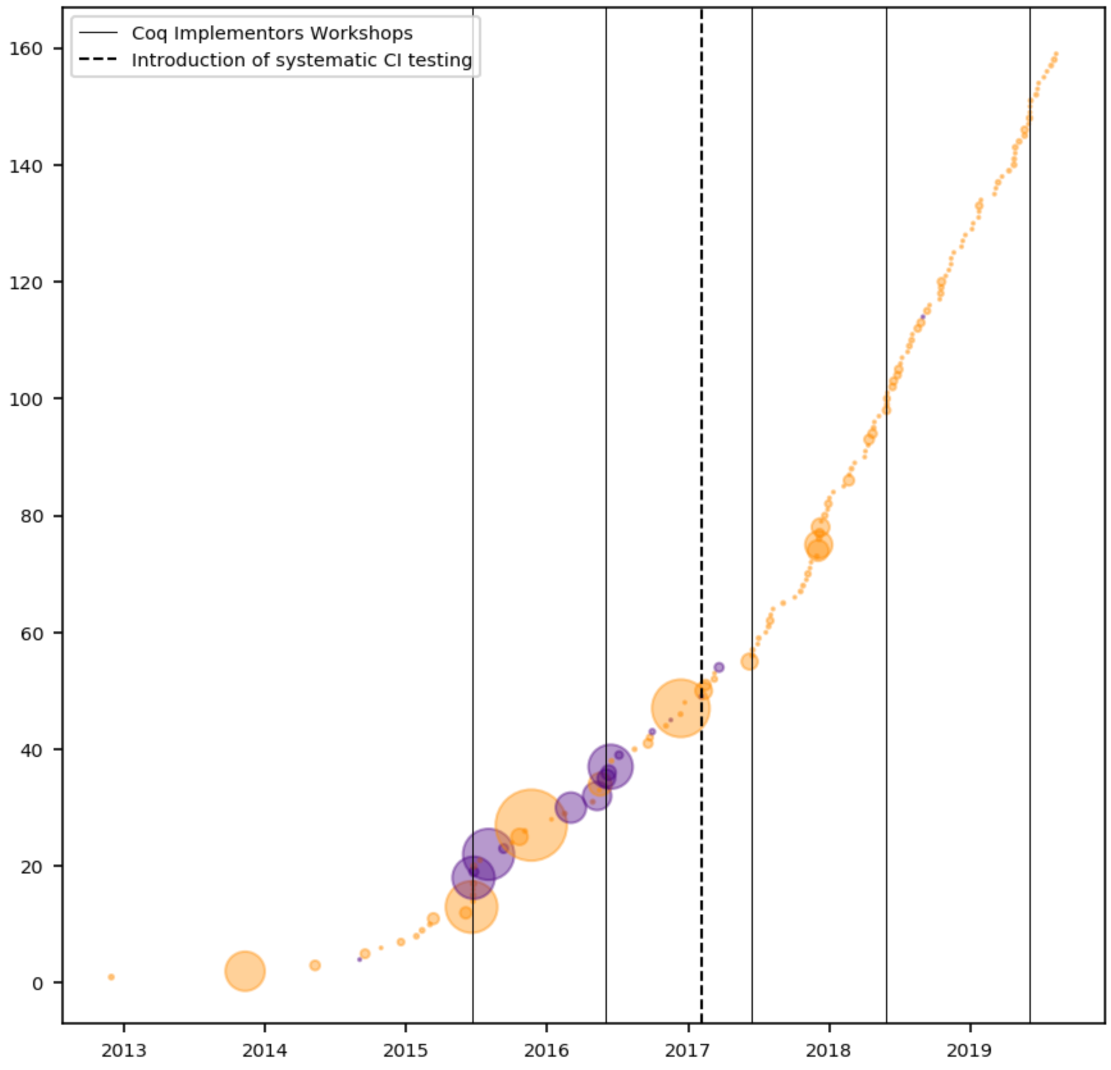


Figure 3.1: Contributors to the Coq GitHub repository. Each dot represents a contributor. Its horizontal position corresponds to the date the contributor opened their first pull request. Its vertical position represents a counter of the number of contributors so far. The area of the dot is proportional to the number of pull requests that the contributor has opened since then. Dark indigo dots represent contributors who already had push access *before* opening their first pull request.

been removed from the dataset, and the developer has been classified as having had push-access.

Figure 3.3 shows the evolution of the number of pull requests opened per day by main developers and by other contributors. It was generated by reusing the analysis pipeline of the bug tracker switch (see Chapter 4).

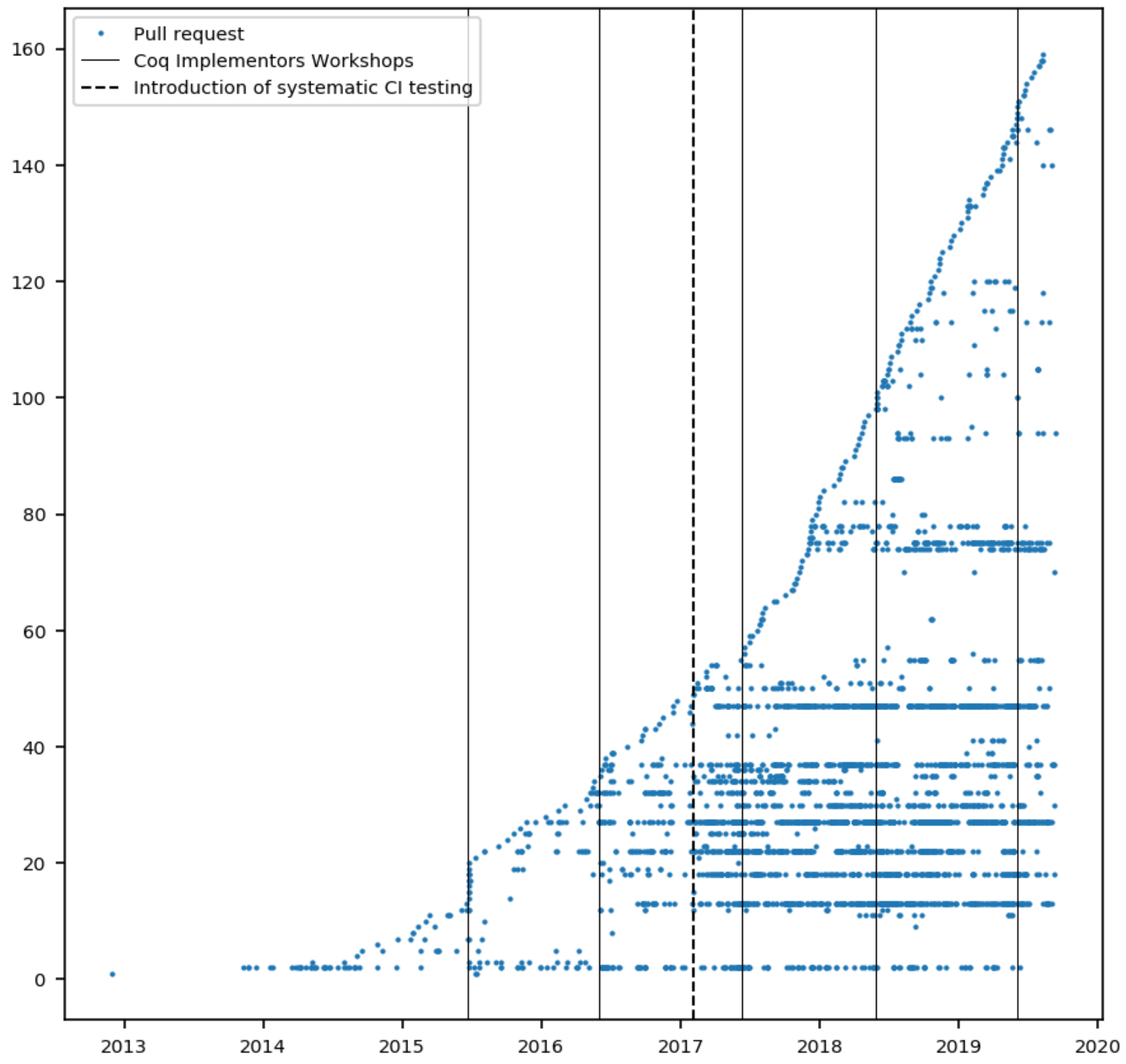


Figure 3.2: Pull requests on the Coq GitHub repository grouped by author. All the pull requests opened by the same contributor are grouped at the same vertical position (and their horizontal position denotes the date when they were opened). Contributors are ordered by when they opened their first pull request.

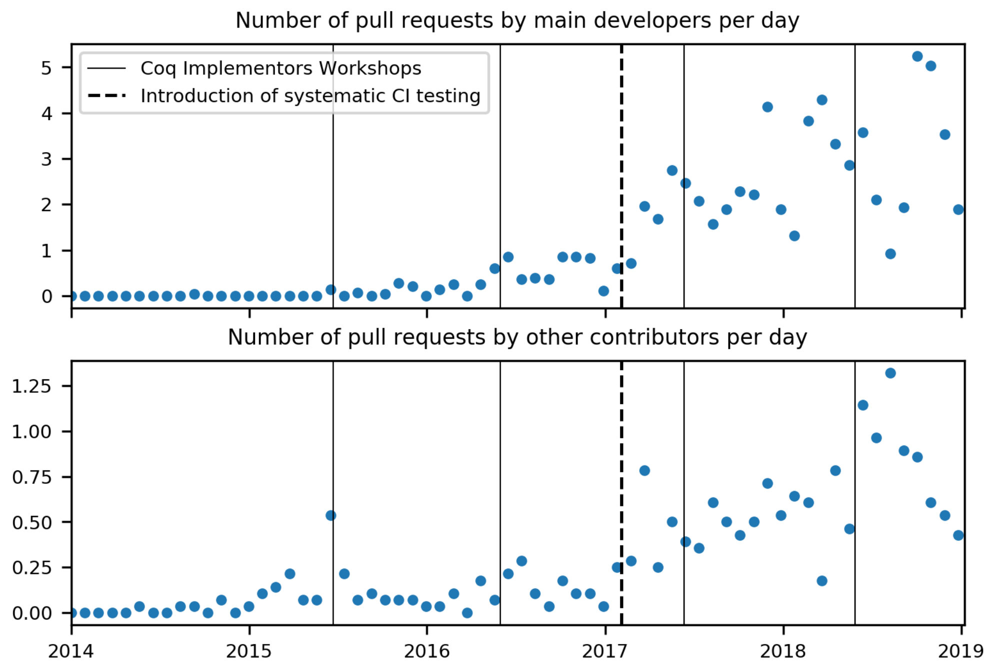


Figure 3.3: Total number of pull requests opened per day by main developers and by other contributors. Each point represents an average over a four-week period. The distinction between the two categories is based on the number of commits, as explained in Section 4.4.2.3.

3.4 How to onboard new contributors?

Core developers are sufficiently many to address by themselves all the issues (bugs and feature requests) that would be worth addressing. Furthermore, core developers frequently leave the project, or become less active, because their focus has changed, and it is important to get new active contributors that can take their place. Finally, the landscape of proof assistants is evolving rapidly, and Coq has both well-established (Isabelle/HOL), and new (the Lean prover) competitors (that are free software as well), thus evolving too slowly could lead Coq to become less successful, as users turn to more modern / shiny solutions. For all these reasons, it seems important to onboard new contributors that can become regular, active developers.

While drive-by contributions do not generally represent a large proportion of the code changes [248], they are also essential to get small fixes, in particular documentation fixes. Indeed, the ones that are most likely to read, and thus discover problems with the documentation, are the users, rather than the developers who feel that they already know the documentation well.

Some actions can be implemented that specifically encourage drive-by contributions, such as a visible “Edit on GitHub” button on each documentation page, or asking to users reporting a bug and proposing a solution whether they would be willing to open a pull request. Some other actions can be taken specifically to retain new contributors, such as providing feedback quickly, and showing appreciation (for instance the reaction system that GitHub introduced in 2016 [38] provides an easy, and non-intrusive[9] way of showing appreciation, that can help newcomers feel welcome[10]).

Code documentation, and contributing guidelines, are essential to new contributors, as was highlighted by Gousios *et al.* [122]. This was frequently reminded by the new contributors themselves, and their pressure and contributions have been an efficient way of improving such documentation. In particular, a good way of getting better internal documentation that will help the next contributors is often to encourage the newcomers who ask questions to write down and share the answers they get.[11]

The contributing guide [50], in particular, is the central piece of documentation aimed at explaining the contribution process to newcomers, and serves as a reference to regular contributors. Its first version was created by an external contributor. I have recently refactored, rewritten, and expanded it, so that it can serve as an exhaustive reference of the state of our current development processes (contributing guides are rarely exhaustive descriptions of the development processes according to Elazhary *et al.* [88]). It is structured in a way that follows the same path as new contributors, from contributions to the ecosystem, to bug reporting and small fixes, to larger contributions and involvement in the merging process [150, 202, 310].

3.4.1 Future work

So far, efforts to bring Coq developers and users closer, such as the creation of the Coq Implementors Workshop (cf. Footnote 8) have been quite successful to transform users into regular contributors. As we will see in Chapter 4, the switch of bug tracker from Bugzilla to GitHub, has also resulted in more external contributors participating in the discussions on issues. In the future, I would like to try to bring development discussions closer to where the users might find them, thanks to the Discourse forum that was launched recently.

Discourse [17] is a new generation, open source, forum software, with a good mailing-list mode. It has been adopted by a great number of open source projects, such as Atom `https://discuss.atom.io/`, Docker `https://forums.docker.com/`, Elm `https://discourse.elm-lang.org/`, Ember.js `https://discuss.emberjs.com/`, OCaml `https://discuss.ocaml.org/`, and Rust `https://users.rust-lang.org/`.

[9]Leaving a reaction on an issue, pull request, or comment does not generate any notification that could lead someone to interrupt their work. Reactions are displayed visibly, but do not take any additional space in a discussion thread.

[10]For some anecdotal evidence, see: `https://gitter.im/coq/coq?at=5b3758513d8f71623d51075b`.

[11]Cf. in particular `https://github.com/coq/coq/pull/961`, where a contributor introduced the first version of the contributing guide, after getting annoyed with something he did not understand, and having discussed it on our Gitter [283] chat; `https://github.com/coq/coq/pull/6557`, where a contributor introduced the test-suite documentation, after asking some questions about it, and being encouraged by myself to start such a document; `https://github.com/coq/coq/pull/7942`, where a contributor introduced a beginners' guide and an index of the developer documentation, after I encouraged him to do so on Gitter.

A user who had used the OCaml instance suggested that we create one for Coq, which I did in the beginning of 2019, after Discourse announced free hosting for open source projects [258], and after having got support from the rest of the development team.

One of the advantages of such a forum over mailing lists,[12] even for users that wish to use it in mailing-list mode, is that it is possible to mute all activity belonging to some categories. This opens the door to creating categories for a subset of users, such as non-English categories. It also makes it possible to move activity from several mailing lists, such as a developers' mailing list and a users' mailing list to distinct categories of a single forum. This is something that Python did by gathering many mailing lists (committers', developers', users' mailing lists) into a single forum `https://discuss.python.org/`.

In their case, they decided to restrict who can post (but not who can read) the committers' category. In the case of the Coq Discourse forum `https://coq.discourse.group/`, the development category is open to all, but has not been used much so far.

I hope to convince the development team to close the developers' mailing list in favor of this forum, as a way to bring development discussions closer to where users can see them. Given the number of communities already using Discourse actively, there might be enough data already available to analyze the effect of such a switch, similarly to what Squire [263] and Vasilescu *et al.* [288] did about Stack Overflow.

3.5 How to improve quality through the use of pull requests?

Because they provide an opportunity for code review, the use of pull requests is largely considered to be associated with a higher level of quality of integrated changes [160]. However, such an effect on quality is not easily measured, and also not automatic as it strongly depends on the quality of the review process itself. Instead of addressing the general question of code quality in this section, I address a more specific question: how to ensure that pull request authors include changes to the documentation, the changelog, and the test-suite, in their pull requests, when such changes are needed.

While it should normally be the role of reviewers to ensure that such changes have been included, it is not always easy for them to remember that they have to check this. To make their job easier, we have introduced two changes: "needs" labels, and pull request templates.

3.5.1 Label categories, the "needs" labels

Labels are quite useful to quickly mark different type of pull requests (or issues), and their current status. Previous work [46] has shown that few projects use labels on their issues (3.25% of non-fork repositories hosted on GitHub), but this result is not surprising given that other

[12]Other advantages being that it is easier to browse and search past questions and answers, and that users do not need to subscribe to a list in order to ask a question.

studies have highlighted the fact that most GitHub projects are single-person projects [157], and rarely use issues at all [29]. Projects where more people are involved are also associated with a higher number of labels.

When a project starts to use more and more labels, the costs may start outweighing the benefits, because participants are not necessarily aware of all the labels that are available, or what they mean. To alleviate this effect, some projects have adopted strategies based on gathering labels in consistent sub-groups. Given that GitHub sorts labels alphabetically, this strategy is particularly efficient when label sub-groups are identified by a common prefix.

After the use of labels in the Coq project had grown to a level where this problem was starting to show up, I introduced two categories ("needs:" and "kind:") and distributed all the preexisting labels between the two. The switch to the GitHub issue tracker, and the migration of preexisting issues (see Chapter 4) led to the addition of new categories ("part:", "resolved:", "platform:"). Overall, the lesson of introducing these categories is that it has allowed the number of labels to grow much further, from 11 labels before the introduction of the categories to 89 today.[13]

The "needs:" category is especially useful for pull request reviewers as a way of reminding what needs to be done before a pull request can be merged. The absence of labels of this category is even checked automatically by the script that is used to merge pull requests (see Section 3.7). This category currently comprises 21 labels, notably a "needs: documentation" label, a "needs: changelog entry" label (initially "needs: CHANGES", before the redesign of the changelog presented in Section 5.5), and a "needs: test-suite update" label.

These categories (the "needs:" category in particular) were inspired by similar categories in the nixpkgs project[14] and have since then inspired the Math-Comp project.[15] Even though label archeology is difficult on GitHub, because GitHub does not keep the history of label renamings and deletions, nor the name of their creators, we can still extract from the GitHub GraphQL API the list of all labels in a project, in the order they were created, their creation date, and the date of their last update. I identified that way that the current naming pattern of labels in the nixpkgs project dates back from November 21–22, 2015, and was introduced by Nicolas Pierron. After I wrote to him, he gave me some pointers,[16] and told me that the inspiration to create categories was Rust. Rust categories are a single letter followed by a dash[17] but Nicolas thought it was important that label categories be understandable by humans (labels can now carry a description but this feature was only added in 2018 [290]), so he introduced longer prefix categories, with an initial number to sort them. The introduction of categories allowed the number of labels of the

[13]8 labels were created following the introduction of the categories; 32 labels were created during the bug tracker migration; 38 labels have been created since then.

[14]See `https://github.com/NixOS/nixpkgs/labels`.

[15]See `https://github.com/math-comp/math-comp/labels`.

[16]E-mail thread in which the introduction of categories was discussed: `https://nixos.org/nix-dev/2015-November/018714.html`. This was not the first time such an idea was proposed: `https://nixos.org/nix-dev/2015-March/016549.html`.

[17]See `https://github.com/rust-lang/rust/labels`.

project to grow from 27 to 114 today.[18]

A further cursory look at the labels of a hundred popular projects leads me to believe that not all popular projects use many labels, but among those that do, most use a similar pattern, where a prefix of the name indicates a category (the separator ":" is quite frequent, but we can also find the separators "/" and "-"). In particular, the Angular.js project has used similar categories, including the "needs:" category, since August 2013.[19] Again a similar growth in the number of labels can be observed.[20] Overall, it would seem that the creation of categories fosters the growth of the number of labels (as we have observed in the Coq, nixpkgs, and Angular.js projects), but this would have to be confirmed through a more in-depth study.

3.5.2 Pull request templates with checklists

Labels are useful for reviewers to mark things that need to be done before merging, but they still have to remember to check that these things were done in the first place. To remind to both pull request authors and reviewers when the pull requests should include a documentation update, a changelog update, or a test-suite update, we have introduced a pull request template including a checklist.

The ability to define a pull request template was added in GitHub in 2016 [31]. From the start they advertised the inclusion of checklists in these templates. Checklists had been introduced three years earlier in GitHub Flavored Markdown [275]. Checklists have been advocated as underused and underappreciated memory aid to increase success in areas like medicine and industry [105], in preparing systematic review papers [216], and during the scientific peer review process [219]. Pull request templates have not yet received much attention from researchers. In 2017, Coelho and Valente [54] showed that very few projects used them. But this was not long after the feature was introduced, and the numbers could have changed since then.

I decided to introduce the first version of this template[21] when I noted, while preparing a release, that a significant number of feature pull requests, which should normally include an update to the changelog, did not. Yet, the value of the template was debated. In particular fears were expressed that it would add bureaucracy that could discourage contribution, and that such template would be inefficient in achieving its objective. While the template that was merged was lighter than the initial proposal, to account for the comments that were made, the fact that its

[18] 6 labels were created automatically when the repository was created on June 4th, 2012; 21 labels were created between July 20th, 2012 and November 16th, 2015; 14 additional labels were created on November 21–22, 2015 by Nicolas Pierron when he introduced the categories; 20 labels were created in just the following year; 53 labels have been created since then.

[19] See `https://github.com/angular/angular.js/labels` and `https://github.com/angular/angular.js/blob/04a570d/TRIAGING.md`.

[20] 14 labels were created between the project's creation date in January 2010 and February 2013; 10 labels were created during the month of August 2013 when the categories were introduced; 48 labels were created in just the following year; and 25 since then.

[21] Cf. `https://github.com/coq/coq/pull/975`.

introduction was controversial makes it all the more important to empirically validate whether it was useful. This is what I do in the next section.

The initial template was merged on December 29th, 2017, and it embedded only two checklist items, reminding to update the documentation, and the changelog. A third item was introduced on June 4th, 2018 to remind to update the test-suite.

3.5.3 Empirical validation

I evaluate the effect of introducing the labels and the templates using the RDD methodology presented in Section 2.3.2. All the details of this evaluation can be found in the corresponding Jupyter notebook [320]. This includes graphs that complement the tables presented here.

3.5.3.1 Data

Data on all the pull requests on the GitHub repository is fetched using the GraphQL API, as presented in Section 2.2.1. For each pull request, its creation date, merged status, labels, number of updated files, and list of updated files (limited to 100 files) are queried. The type of updated file is matched based on the file path: pull requests updating the changelog are the ones listing the file `CHANGES`; pull requests updating the documentation are the ones listing files under the `doc/` directory; pull requests updating the test-suite are the ones listing files under the `test-suite/` directory.

Pull requests that only updated the changelog (1%), or only updated the documentation (6%), or only updated the test-suite (3%) are excluded from this dataset, because the goal is to measure the presence of documentation, changelog, and tests in pull requests that are not primarily about updating such components. Pull requests that updated more than 100 files (2%) are excluded from the dataset because the list of updated files that has been queried is truncated to the first 100.

3.5.3.2 Effect of the labels

I start by evaluating the effect of "needs: documentation" and "needs: CHANGES" labels. The "needs: documentation" label has been introduced 142 days before the pull request template (that included a checklist item reminding about updating the documentation). To avoid considering a dataset that includes more than one discontinuity, I use a bandwidth of 142 days on each side of the cutoff (which is the date the "needs: documentation" label was introduced). Similarly, because the "needs: CHANGES" label was introduced 82 days after the pull request template, I use a bandwidth of 82 days to evaluate its effect.

In both cases, I do not obtain statistical significance for the coefficients of interests γ_2 (representing the jump immediately after the cutoff), and γ_3 (representing the change in slope after the cutoff). The estimated values for γ_2 are even negative (see Table 3.1). An absence of

	Proportion of pull requests including documentation (*cutoff:* introduction of the “needs: documentation” label)	Proportion of pull requests including a changelog entry (*cutoff:* introduction of the “needs: CHANGES” label)
γ_2 *(After event)*	-0.061 (0.039)	-0.017 (0.046)
γ_3 *(Relative date × After event)*	-0.0005 (0.0004)	0.0008 (0.0011)
γ_1 *(Relative date)*	0.0005 (0.0003)	-0.0009 (0.0009)
γ_0 *(Constant)*	**0.11***** (0.030)	**0.10**** (0.035)
Observation number	868	615

Table 3.1: Estimated effects of the introduction of the “needs: documentation” label on the presence of documentation, and of the “needs: CHANGES” label on the presence of a changelog entry. Statistically significant results are in bold (* means $p < 0.05$, ** means $p < 0.01$, *** means $p < 0.001$). Standard errors are in parentheses.

measured statistically significant effect does not mean that there was no effect, but that the potential effect is too low to measure. This is not surprising because, while these labels are helpful for reviewers to communicate what needs to be done, they still have to remember about those things to use the labels. And when the labels did not exist, if the reviewers remembered about those things, they had other ways to communicate them.

3.5.3.3 Effect of the template

Given the lack of statistical significance, and the low, and even negative coefficients that were estimated for γ_2 on these two RDDs, I consider safe to include a larger bandwidth of 150 days, on each side of the cutoff, when estimating the effect of the introduction of the pull request template. This means that the dates of introduction of the labels are included in the considered period, but they are not discontinuities that would influence the results of the RDD.

Table 3.2 gives the results of the RDDs. We can see that we get a statistically significant jump on both the presence of documentation and of a changelog entry. The estimations tell that the proportion of pull requests that include documentation more than doubled after the introduction of the template, from about 6% to about 16%, and that the proportion of pull requests that update the changelog more than quadrupled after the introduction of the template, from about 4% to about 17%.

Next, I evaluate the effect of adding the checklist item reminding about updating the test-suite in the pull request template. The “needs: test-suite update” label and the checklist item in the template were introduced within one week of each other, which defeats any hope of

	Proportion of pull requests including documentation	Proportion of pull requests including a changelog entry
γ_2 *(After event)*	**0.095*** (0.038)	**0.13**** (0.040)
γ_3 *(Relative date × After event)*	-0.0002 (0.0004)	-0.0006 (0.0005)
γ_1 *(Relative date)*	0.0000 (0.0003)	-0.0002 (0.0003)
γ_0 *(Constant)*	**0.062**** (0.022)	0.036 (0.021)
Observation number	1009	1009

Table 3.2: Estimated effects of the introduction of the pull request template on the presence of documentation and of a changelog entry. Statistically significant results are in bold (* means $p < 0.05$, ** means $p < 0.01$, *** means $p < 0.001$). Standard errors are in parentheses.

estimating separately the effects of the two. However, given our previous results on the "needs: documentation" and "needs: CHANGES" labels, I am going to consider that the "needs: test-suite update" label probably had no effect either, and I am going to attribute any (combined) effect that I measure to the modification of the pull request template.

Once again, I consider a bandwidth of 150 days, around a cutoff corresponding to the introduction of the checklist item in the pull request template. Note that this event occurred more than 150 days after the pull request template was introduced, which means that there was a pull request template during the whole period considered, and that the only change was whether this template included a checklist item reminding about the test-suite, or not.

Table 3.3 gives the results of the RDD. We can see a statistically significant jump after the introduction of the checklist item. The estimated effect is that the proportion of pull requests including an update to the test-suite more than doubled, from about 14%, to about 29%.

As robustness checks, I estimate RDDs on larger bandwidths, of 450 days on each side of the cutoff (which is the largest possible bandwidth given the currently available data), with both linear and quadratic specifications. All the results continue to hold with similar estimated coefficients, and statistical significance.

I also estimate RDDs on merged pull requests only, and once again obtain similar, statistically significant results.

3.5.3.4 Heterogeneous effects

Finally, I estimate RDDs on specific types of pull requests (as determined by their "kind:" label): features, and bug fixes. Given that this reduces the number of observations (only 6% of pull requests are features, and 30% are bug fixes), statistically significant effects can only be detected if the estimated effect is sufficiently large to compensate.

	Proportion of pull requests including tests
γ_2 *(After event)*	**0.15**** (0.048)
γ_3 *(Relative date × After event)*	0.0005 (0.0006)
γ_1 *(Relative date)*	**-0.0010*** (0.0004)
γ_0 *(Constant)*	**0.14***** (0.032)
Observation number	1010

Table 3.3: Estimated effects of the introduction of a checklist item reminding about updating the test-suite on the presence of tests. Statistically significant results are in bold (* means $p < 0.05$, ** means $p < 0.01$, *** means $p < 0.001$). Standard errors are in parentheses.

Features Pull requests marked as features are likely to warrant a changelog entry, an update to the documentation, and new tests. However, we can only detect a statistically significant effect of the pull request template on the presence of a changelog entry (see Table 3.4). The estimated jump in the proportion of feature pull requests that include a changelog update is of more than 60%. This result continues to hold in the robustness checks.

For the documentation and the tests, the estimated coefficients γ_2 are positive, but not sufficiently high compared to the estimated coefficients γ_0 and the standard errors. It is probable that the template had a positive effect on feature pull requests, but that we fail to detect it with statistical significance because the proportion of feature pull requests including documentation and tests was already relatively high *in the absence of the template*.

On the contrary, we notice a large effect on the presence of a changelog entry, because the proportion of feature pull requests including one, before the introduction of the template, was quite low. Given that my initial motivation for introducing the template was the high number of feature pull requests missing a changelog entry, this is not surprising, and confirms that this was an appropriate course of action to address the problem.

Bug fixes Pull requests fixing bugs are likely to warrant the introduction of a new test. The results of the RDD on bug fix pull requests show that about 39% of them already included tests at the time the checklist item was introduced in the template, and do not show any significant effect of this introduction.

3.5.3.5 Threats to validity

The main threat to consider, when performing causality analysis using RDD, is that there could be other discontinuities, close to the cutoff, that could account partially for the observed

	Proportion of PRs incl. documentation (*cutoff:* intro. of the template)	Proportion of PRs incl. a changelog entry (*cutoff:* intro. of the template)	Proportion of PRs incl. tests (*cutoff:* intro. of the checklist item)
γ_2 *(After event)*	0.15 (0.24)	**0.63**** (0.23)	0.35 (0.26)
γ_3 *(Rel. date × After event)*	-0.0003 (0.0032)	-0.0005 (0.0031)	0.0025 (0.0027)
γ_1 *(Rel. date)*	-0.0007 (0.0021)	-0.0020 (0.0021)	-0.0033 (0.0018)
γ_0 *(Constant)*	0.27 (0.16)	0.15 (0.16)	0.21 (0.16)
Observation number	69	69	70

Table 3.4: Estimated effects of the introduction of the pull request template (resp. the checklist item reminding about tests in the pull request template) on the presence of documentation and a changelog entry (resp. on the presence of tests) in feature pull requests. Statistically significant results are in bold (* means $p < 0.05$, ** means $p < 0.01$, *** means $p < 0.001$). Standard errors are in parentheses.

effects. In this case, I have identified such discontinuities corresponding to the introduction of labels, but I have discarded them based on the results I got for the "needs: documentation" and "needs: CHANGES" labels. However, for the effect of the checklist item on the presence of tests, there is no way to totally discard the hypothesis that the introduction of the label was partially, or entirely responsible, for the effect that was observed.

Other events that could have impacted the results include new releases, and developers' vacation. Following a release, we could imagine that the type of pull requests that are opened changes, and therefore the need for documentation, tests, or changelog entries as well. Nevertheless, I was able to show an effect on the changelog for feature pull requests only, which means that at least the result on the changelog is not a consequence of a variation in the type of pull requests. Furthermore, the inclusion of larger bandwidths in the robustness checks allows to include several such events on both sides of the cutoff, which reduces the risk that they impacted the measured effects.

Another threat to consider is that the simple linear functional form chosen for the regression could not accurately model the evolution of the outcome variable over time. That is why I also consider quadratic forms in robustness checks. But, most importantly, I plot the data (see the Jupyter notebook [320]), which allows to visually verify that an alternative, better suited, functional form, is not being neglected.

Finally, given the limited scope of this evaluation, it is not guaranteed that the results will generalize to other projects. Some characteristics specific to the Coq project could have been key to the successful use of a checklist in the pull request template. In particular, in projects where a strong reviewing culture has been in place for longer, it is possible that such checklist would have little or no effect. However, for projects whose developers notice that contributors frequently forget about something, this first empirical result can be a basis to recommend the use of a template with a checklist.

Reproducing these results on other projects would be really useful to be able to draw generic recommendations. However, doing such an evaluation on a large scale is not likely to be possible given the wide variety of pull request templates that projects can use, and the wide variety of needs for a checklist they can express.

3.6 How to test compatibility with reverse dependencies in CI?

3.6.1 Introduction of continuous integration in Coq

The Coq sources have included a test-suite from at least the beginning of the current repository, in 1999. This test-suite includes tests of success and failure, and non-regression tests added when bugs are fixed. In the previous, push-based, development model, developers were used to running this test-suite before committing. In the new, pull-based development model, it is important that this test-suite is run as part of an automated, continuous integration (CI) process on each pull request, so that reviewers can review a pull request without having to run the tests themselves, and external contributors can understand the kind of expectations their pull request must meet [122], and even fix the pull request when tests are failing, sometimes before the reviewers had any chance to make a comment.

Systematic CI was introduced in the Coq project in the beginning of 2017, initially using Travis CI [99] (that Hilton *et al.* reported in 2016 to be by far the most popular CI provider [133]), after a previous experience with a Jenkins [161] server that had never been successfully set up to run on all pull requests and report results back.

Widder *et al.* [303] suggested that projects that invested significant effort in their CI configuration are less likely to abandon Travis, but the Coq project provides an anecdotal counter-example. Most of the complexity of our CI system is either in shell scripts and build tools that can be reused from one CI service to the next, or in the conceptual organization of the configuration file, that can be easily replicated in the next service (as long as certain features are supported). Not counting our initial experience with Jenkins, the Coq project tested (successively or in parallel) five different CI providers in a two-year period:[22] Travis CI [99], AppVeyor [15], CircleCI [51], GitLab CI [118], and Azure pipelines [196] (more on this in Section 3.6.4).

[22]One way of retracing the history of the various services is to look at the badges that were successively added and removed in the commit history of the README: `https://github.com/coq/coq/commits/master/README.md`.

3.6.2 Compatibility testing with external reverse dependencies

Coq is a programming language, with a strong type system, and a complex and underspecified machinery for building proofs called "tactics" [59]. Many projects depend on Coq, either because they are written in Coq, or because they are Coq plugins, written in OCaml, and interfacing with the Coq API.

When producing a new major release, it is important to be able to assess the level of compatibility breaking changes that projects will encounter while attempting to update. Hora *et al.* [136] note that, in the context of large software ecosystems, "it is hard for developers to predict the real impact of API evolution". Ma *et al.* [180] identified a demand for tools supporting effective cross-project testing. Many researchers have worked on tools to automatically adapt to API evolution, including in the context of Coq [245]. Sometimes, the developers themselves can provide simple scripts [126], or complex translators [22], to adapt to some specific changes. However, while tooling can help an ecosystem move forward, it will always have limitations, and being able to evaluate *a priori* the amount and difficulty of required adaptation is very important.

High levels of compatibility breaking changes, or breaking changes with no instructions or unclear / untested instructions on how to port existing code, are not acceptable because they can cause projects to stay stuck on an outdated version. This is what happened with the Coq 8.5 release: some projects took years to migrate. In 2019, some people still depended on Coq 8.4 for their work,[23] three years after version 8.5 had been released, and while four additional major versions had been released since then. Some projects were rewritten from scratch rather than being ported.[24]

And yet, Coq already had a compatibility benchmark at that time. It was composed of a set of Coq projects called "contribs" (for user contributions) which had been the primary distribution channel of Coq packages from the beginning, until a package manager started to be used (see Section 6.5.1). However, with the increase of the number of Coq users, many projects were missing from this set (often the ones under heavy development, and therefore likely to be using the most recent techniques or features). Furthermore, developers were not used to check that they did not break any package before pushing their commits. Thus, breaking changes would be discovered much later, and most often, the tested packages would be adapted rather than the change reverted.

The introduction of CI was an opportunity to put in place a new system, where pull requests are checked for compatibility with a selection of reverse dependencies *before* they are merged. Another independent change that happened simultaneously was to revise the set of tested packages, and to ensure that we would keep testing their official development versions instead of

[23] See an example at: `https://github.com/NixOS/nixpkgs/issues/54556`.

[24] The Bedrock project (`https://github.com/mit-plv/bedrock`) was partially ported but still depended on an 8.4 compatibility option which was eventually removed. The project was abandoned in favor of a new project called Bedrock2 (`https://github.com/mit-plv/bedrock2`).

forks.[25]

With Emilio Gallego Arias, we designed a simple system called “overlays” which allows the author of a pull request to test it with patched versions of the reverse dependencies that are included in CI.[26] The principle is to create a file in a designated directory[27] that overwrites the defined location of the source of a tested project, but only under certain conditions (only for a specific branch name or pull request number, as determined through specific CI environment variables). Thus, these overlay files, even if they are merged with a pull request, will not change what is tested on the main development branch (master). To avoid breaking the build on master, the integrator must ensure that “all the overlays have been merged upstream” (meaning that the patched versions of the projects have been integrated, usually through dedicated pull requests, by the maintainers of the reverse dependencies that we test). This can be done early on when these patches are backward-compatible (meaning that the patched versions also build with earlier versions of Coq), or must be done in sync with the change in Coq when they are not backward-compatible. Our merging script (cf. Section 3.7) detects the presence of overlays and reminds the integrator about this requirement.

3.6.3 Compatibility policy, and CI inclusion policy

3.6.3.1 Compatibility policy

Following the experience of the Coq 8.5 release, a clear compatibility policy has been designed to avoid reproducing the same issues. The main ideas are very basic: it should always be possible for a project written in Coq to be compatible with two successive major versions, features must be deprecated in one major version before they can be removed in the next, Coq developers must have a good estimate of the effort it takes to fix a project with respect to a given change, and finally and most importantly, breaking changes must be clearly documented with clear recommendations explaining how to fix a project.

In practice, this means that new behaviors are often gated behind options (this was already often the case in Coq 8.5 and before) that are turned on by default when they are considered to be better for new projects, and are deprecated from the start when the developers want to be able to get rid of the underlying code as soon as possible. Fixing the external projects that are tested in CI is the responsibility of pull request authors. Therefore, they are well-aware of the effort it takes, and they can document it precisely in the changelog. Furthermore, they are nudged towards adapting their pull request to avoid breaking compatibility, when preparing patched versions of all the broken projects would be too much effort.

[25]See https://github.com/coq/coq/pull/421.

[26]See https://github.com/coq/coq/pull/456, https://github.com/coq/coq/pull/477, and https://github.com/coq/coq/pull/779.

[27]The use of a directory, rather than a single file, to gather all overlays, is a simple trick to avoid merge conflicts that we will use again for the changelog in Section 5.5

Removing a deprecated compatibility option or deprecated feature can be reconsidered upon discovering, thanks to CI, that many projects still depend on it, and that it would be a lot of work to fix them all. CI can also be used to assess the consequences of removing a feature before taking a decision to deprecate it.

Finally, because it must be possible for the projects written in Coq to stay compatible with two successive versions of Coq, it means that overlays should be backward-compatible, and thus may be merged before the corresponding Coq pull request.

On the other hand, the Coq API is far from stable, and applying a similar compatibility policy about it would create too much constraints preventing the evolution of Coq. That is why the policy regarding plugins is simply to encourage all plugin authors to submit their plugins for inclusion in CI, so that Coq developers can continuously fix them as the Coq API evolves. This compatibility policy for plugins is somewhat close to the one of the Linux kernel for drivers [170], which it is inspired from. The main difference, however, is that most plugins are not developed in the same (git) tree.

3.6.3.2 CI inclusion policy

Our CI inclusion policy has been formalized as a document [101] but it is not yet fully stable. The main ideas are that the projects that wish to be included in CI have a public git repository from which we can fetch the sources, and to which we can easily submit patches, that a branch in this repository (ideally master) is kept compatible with Coq's master branch, and that the maintainer of this repository is responsive (or grants push-access to the Coq developers).

Some questions are still debated: for instance, we want an easy process for submitting patches, but does this mean that projects should be hosted on social platforms such as GitHub (.com), GitLab (.com), or Bitbucket, and that alternative platforms like Inria's GitLab (`https://gitlab.inria.fr/`) which restrict who can submit a pull request should be banned? Should the use of CI in the tested project be mandatory (or is it fine if the maintainers make sure they do not break compatibility with Coq master by other means)?

One point that is still absent from our inclusion policy is what kind of projects we accept. Currently we test a total of 33 external libraries (i.e. projects written in Coq), see Table 3.6 and 3.7, and 12 external plugins (i.e. projects interfacing with the Coq ML API), see Table 3.5, adding to a total of 2,157,883 lines of Coq (Coq LoC), plus 33,559 lines of OCaml (ML LoC) in the external plugins.[28] While we strongly encourage any plugin author to submit their plugins for inclusion because of the absence of a stable API (even unreleased plugins under heavy development), it is likely that there will be limits to the number of libraries we accept, and we will likely favor the most used ones, and the ones that are most useful to test features that would otherwise receive little testing.

[28]This is in addition to the 181,611 Coq LoC in Coq's standard library and internal plugins, 83,868 Coq LoC in Coq's test-suite, 54,496 ML LoC in Coq's internal plugins, and 182,531 ML LoC in Coq's implementation.

Another crucial point that is still absent from our inclusion policy, but will have to be addressed in the future, is the way projects can manage their dependencies. As of today, four different methods are used by various projects:

- Some projects declare their dependencies in our CI infrastructure, and are built with the version of these dependencies that we happen to be testing. This is the best solution to allow reuse of the build artifacts: if the same version of a library is a dependency of multiple projects, it is possible to build it only once.
- Then we have a family of projects that pin their dependencies in opam [107] files. This is a quite complex setup because it requires to fetch the last project in the dependency chain to know which version of the previous project in the dependency chain to fetch.
- Some projects vendor their dependencies using git submodules. This is a simpler setup for project maintainers because they can add / remove dependencies without having to update the CI configuration of Coq. However, it can be a bit harder for Coq developers to track when this is the project or a dependency that needs to be fixed.
- Finally, we have a family of projects that vendor their dependencies by copying the code of the dependency in their repository. This is also more complex to manage for Coq developers, because it is tempting for them to prepare a patch to the vendored library and to submit it to the project doing the vendoring. However, the maintainers of these projects generally prefer the patch to be submitted to the library upstream, so that they can update their copy to match the one from upstream.

To simplify the experience of the contributors to the Coq repository that need to prepare overlays when submitting a breaking change, we are likely going to require that projects that are included in CI have a uniform way of managing their dependencies. The best solution from the point of view of the Coq development is probably to impose that dependencies are declared in Coq's CI configuration, and that for each shared dependency, we can define a single version which all the tested projects are compatible with.

Tables 3.5, 3.6, and 3.7 list all the external projects that are currently tested in Coq's CI. For each project, they give the number of lines of Coq (Coq LoC), and for the plugins they give the number of lines of OCaml (ML LoC) as well. The URL of repositories is provided, except for the GitHub repositories, which are the majority, for which only full names are provided (thus the URL is obtained by prepending "github.com/"). Information on the dependency relations is also displayed.

Note regarding Table 3.5 The Fiat Parsers sources are part of a larger repository, and OCaml sources are duplicated to maintain compatibility with multiple versions. Therefore, I have

Name	GitHub full name	ML LoC	Coq LoC	Dependency of
AAC Tactics	coq-community/aac-tactics	2726	1435	
Bignums	coq/bignums	935	11390	Color, Math-Classes
Coq-Elpi	gares/coq-elpi	726	333	
Coq-Dpdgraph	Karmaki/coq-dpdgraph	412	1421	
Coq-Hammer	lukaszcz/coqhammer	5219	2972	
Equations	mattam82/Coq-Equations	11538	18499	
Fiat Parsers	mit-plv/fiat	80*	42609*	
Mtac2	Mtac2/Mtac2	2992	11524	
Paramcoq	coq-community/paramcoq	1671	885	
QuickChick	QuickChick/QuickChick	5611	19092	
Relation Algebra	chdoc/relation-algebra	386	8617	
Unicoq	Unicoq/Unicoq	1263	118	Mtac2

Table 3.5: The external plugins tested in Coq's CI. They are all declared (no implicit dependencies). Few have other tested projects depending on them, and few have external dependencies.

restricted the LoC counting to the sub-folders `src/Common/Tactics` and `src/Parsers`, and to only one copy of the OCaml files.

Notes regarding Table 3.6 The CompCert LoC are counted after excluding the `flocq/` and `MenhirLib/` sub-folders, which are vendored dependencies. The Math-Comp LoC are counted after excluding the `mathcomp/ssreflect/` sub-folder, which corresponds to the SSReflect LoC. VST LoC are counted after excluding the `compcert/` and `compcert_new/` sub-folders, which are two different copies of the same vendored dependency. Note that VST has other dependencies declared as sub-modules, but these are not needed in the default target that is tested in Coq's CI. Software Foundations LoC only count the content of the three first parts (Logical Foundations, Programming Language Foundations, and Verified Functional Algorithms), which are the ones that are actually tested in CI.

Note regarding Table 3.7 BBV (Bedrock Bit Vectors) is a dependency of a legacy version of Fiat-Crypto that is still tested for its unique use of Coq's notation system.

3.6.4 Assessment

The switch to *a priori* testing has been incredibly successful in implementing our compatibility policy.

First, the "no stable API" policy has left developers free to improve the API and conduct refactorings as they saw fit. In practice, fixing the external plugins is not much more work compared to fixing the internal plugins, and the main components of the Coq implementation. It has also given plugin developers a chance to get closer to the Coq development team, and to ask

Name	GitHub full name or URL	Coq LoC	Dependency of
Argosy	mit-pdos/argosy	6095	
Bedrock2	mit-plv/bedrock2	22533	
Coquelicot	gitlab.inria.fr/coquelicot/coquelicot	43408	
Color	fblanqui/color	108184	
CompCert	AbsInt/CompCert	152036*	VST (copy)
Corn	coq-community/corn	121193	
Cross-Crypto	mit-plv/cross-crypto	8239	
Ext-lib	coq-ext-lib/coq-ext-lib	8212	QuickChick, Simple IO
PCM	imdea-software/fcsl-pcm	5739	
Fiat-Crypto	mit-plv/fiat-crypto	65021	
Flocq	gitlab.inria.fr/flocq/flocq	68766	CompCert (copy)
GeoCoq	GeoCoq/GeoCoq	134415	
HoTT	HoTT/HoTT	52728	
Iris	gitlab.mpi-sws.org/iris/iris	40055	Lambda-Rust
Lambda-Rust	gitlab.mpi-sws.org/iris/lambda-rust	17791	
Math-Classes	coq-community/math-classes	15549	Corn
Math-Comp	math-comp/math-comp	92969*	
Simple IO	Lysxia/coq-simple-io	859	QuickChick
Software Foundations	softwarefoundations.cis.upenn.edu	62152*	
SSReflect (part of Math-Comp)	math-comp/math-comp	19579*	Coquelicot, PCM, QuickChick, Std++
Stdlib2	coq/stdlib2	1934	
Std++	gitlab.mpi-sws.org/iris/stdpp	17427	Iris
UniMath	UniMath/UniMath	176053	
Verdi-Raft	uwplse/verdi-raft	46491	
VST	PrincetonUniversity/VST	615953*	

Table 3.6: The Coq libraries (Coq projects which do not interface with the Coq API) which are *explicitly* tested in CI (excluding implicit dependencies). Currently, the vendored dependencies are built multiple times.

Name	**GitHub full name or URL**	**Coq LoC**	**Dependency of**
Array	tchajed/coq-array	533	Argosy (submodule)
Bedrock Bit Vectors	mit-plv/bbv	7920	Fiat-Crypto* (submodule)
Classes	tchajed/coq-classes	508	Argosy, Array (submodules)
Coqprime	thery/coqprime	64834	Fiat-Crypto (submodule)
Coqutil	mit-plv/coqutil	3874	Bedrock2 (submodule)
Kami	mit-plv/kami	47860	Bedrock2 (submodule)
MenhirLib	fpottier/coq-menhirlib	2808	CompCert (copy)
Risc-V (in Coq)	mit-plv/riscv-coq	6920	Bedrock2 (submodule)
Tactical	tchajed/coq-tactical	350	Argosy (submodule)

Table 3.7: The Coq libraries that are tested in CI because they are implicit dependencies (vendored through copying or submodules) of projects that are explicitly tested.

for help when they needed (which is frequently given that the current documentation of the API is far from sufficient). Communication between plugin developers and Coq developers has mostly occurred through GitHub, the Gitter [283] chatroom, and in person during Coq Implementors Workshops (cf. Footnote 8).

Besides, the inclusion of many external projects has given much more confidence when assessing breaking changes. It has also allowed developers to discover bugs in pull requests earlier.

To assess how frequently CI gets broken in pull requests, and what type of breakage this is, I fetched data (cf. the corresponding notebook [318]) on the pull requests that were opened in the last 8 months (the pull requests that were opened after I introduced a naming scheme that distinguishes between plugin and library builds[29]). For each pull request, I request the list of pushed commits, and from the pull request timeline, I retrieve the force-push events,[30] and for each of these commits and events, the results of CI builds. I focus only on the builds originating from GitLab CI, because this is the CI provider that has been used lately to test compatibility with external projects. In parallel, Travis CI and AppVeyor, then Azure pipelines were used to test the build on macOS / Windows.

The proportion of failed builds is estimated to 53%. The proportion of pull requests whose first build is successful is 51%. About 17% of the build failures are due to Coq's compilation being broken. Among the rest of the failures, 30% are due to a breakage in the test-suite, 57% are due to a broken plugin, and 35% are due to a broken library (note that some build failures are counted in multiple categories). About half of the time when the test-suite is broken, no external library is broken simultaneously. A cursory look shows that this mostly corresponds to failing

[29]See `https://github.com/coq/coq/pull/9278`.

[30]In the Coq project, we generally recommend contributors to rebase and force-push when fixing a pull request, to avoid merging broken commits. GitHub introduced, in November of last year [114], a timeline event when a pull request is force-pushed, which is very convenient to have a global view of the pull request history.

output tests (some command produces a different output but this does not change the semantics). On the other hand, more than half of the time a library breaks, no test in the test-suite breaks simultaneously. This shows that the test-suite has some room for improvement, and that in the meantime, testing external projects is quite efficient to make sure that there are no accidental compatibility breakages.

I also estimated the frequency of spurious failures from the proportion of pull requests that were merged despite a failing test. This represents about 18% of the builds, which is quite a high number (and it may be an under-estimation because spurious builds may also be restarted when the integrator is not sure whether they are spurious). Out of these spurious failures, 45% are due to broken plugins: this can be because the maintainer of the plugin broke the build, or because of a synchronization issue following the integration of a pull request with overlays (this is further confirmed by the fact that more than 30% of the time, two or more plugins were broken simultaneously). 28% of these spurious failures are due to broken libraries: this is most likely because the maintainer broke the build (this is further confirmed by the fact that more than 85% of the time, a single library was broken at once).

Technically, the unusual size of our CI has created high constraints that have led us to discover the limits of the various CI providers.

Our first provider was Travis CI [99]. Travis CI gives four parallel jobs for free to any open source project. At the time, it did not have native support for artifact sharing between jobs of successive stages (it could be done with an external service, but not securely on pull request builds). The unfortunate implication was that we ended up building Coq more than 20 times in parallel, to be able to test the build of various external projects afterwards. We got to a point where the total sequential build time was close to 11 hours, and the parallel time was frequently above 2 hours and a half.[31] During busy periods, such as when a release was upcoming, pull requests could end up waiting in the build queue for several days. Fortunately, pull requests authors could activate Travis on their own forks which would allow them to get results faster (without waiting in the queue). During busy periods, the integrator would ask for a link to a successful build on the author's fork rather than waiting for CI to complete on the main repository.

Our second provider (not counting AppVeyor [15] which we used specifically for Windows testing) was Circle CI [51].[32] Circle CI had native support for artifact sharing between jobs, and a very convenient DAG-based workflow.[33] This allowed not only to build Coq only once, but also to build some shared dependencies of external projects only once. Another nice feature of Circle CI was that it reported the build status of each job (instead of a global result), which allowed to see quickly where failures were coming from. Unfortunately, the limit on the number of parallel jobs was shared among all forks belonging to organization members, which led to even faster resource exhaustion, without the workaround that we used with Travis. Because of this, we never

[31] See for instance `https://travis-ci.org/coq/coq/builds/313272872`.

[32] Introduced in `https://github.com/coq/coq/pull/6400` by Gaëtan Gilbert and Arnaud Spiwack.

[33] Jobs depending on other jobs can start as soon as the dependency has completed.

really abandoned Travis while we were testing Circle CI, and we ended up looking for a better alternative.

The solution was to use GitLab CI [118]. Gaëtan Gilbert had started to experiment with it very early.[34] However, it had not been adopted because this solution was not designed to run with GitHub repositories. GitLab added CI support for GitHub repositories in March 2018 [306]. Unfortunately, this only meant mirroring the repository, running CI, and reporting results on GitHub: no support was provided for testing pull requests originating from forks.[35] To solve this issue, I introduced a bot (see Section 2.5) whose job is to create a branch on our GitLab mirror whenever a pull request is opened, to force-push to this branch every time the pull request is updated, and to delete it when the pull request is closed. Furthermore, the bot reports status checks back, instead of leaving that task to GitLab. This allows to get more features such as detailed reporting of failed builds (inspired by CircleCI), testing an automatic merge commit of a pull request with its base branch (inspired by Travis CI), and direct links to HTML artifacts allowing to preview the documentation when it was updated by a pull request.

GitLab CI has native support for artifact sharing between successive stages, and has added DAG-based pipelines very recently [172].[36] It also provides unlimited parallel jobs for free. In practice, the number of runners that are available varies from one to many depending on the time of the day. So even if builds are queuing up at some point, the queue eventually gets cleared at night. Furthermore, we can easily set up servers to become additional runners, which we have done with two machines with many cores that are provided by Inria. Thanks to them we never go down below about 20 parallel jobs.

Since then, the Math-Comp project has adopted a similar testing infrastructure (reusing the bot to be able to run on GitLab CI), but with a slightly different implementation of overlays. The implementation is based on updating the CI configuration file itself [185]. For small projects, another, and even simpler strategy, would be to change the location of the sources of a tested reverse dependency directly where it is defined, and to revert the change as soon as the corresponding patch has been integrated in the reverse dependency.

3.6.5 Related work

The closest effort that this compares to is the Scala Community Builds project [274]. The goals are comparable: detecting regressions sooner, confidently assessing the impact of a proposed change.

In terms of size, the number of tested lines of code is close (2.2 million LoC from external projects in Coq's CI, 3.6 million in Scala Community Builds —as of August 2019), although the

[34]See `https://github.com/coq/coq/pull/687`.

[35]To this day, this is still a very popular but unimplemented feature request: `https://gitlab.com/gitlab-org/gitlab/issues/5667`.

[36]This was a long awaited feature, and we have used it from day one of its release: `https://github.com/coq/coq/pull/10686`.

number of tested projects is much larger in Scala Community Builds (199), than in Coq's CI (45). In terms of total runtime, Scala Community Builds take about 15 hours to run, compared to over 21 hours of total sequential time of Coq's CI (see Section 3.6.6.2).

However, the main difference is that Scala Community Builds are mostly run *a posteriori* (they are run *a priori* on demand[37]) while Coq's CI is run systematically before pull request integration, which avoids introducing regressions.

The main advantage of catching regressions before introducing them in the main branch is that the burden of fixing them does not fall on the same people. When a regression is found in a pull request, its author is likely to make efforts to fix it, because the pull request will not get integrated until the regression gets fixed. On the other hand, when a regression is found after a pull request has been merged, even if the commit introducing it is clearly identified, there is no way (in an open source project where all work is voluntary) to force its author to fix it. In practice, the author may announce intent to look into it, but has no reason to hurry. Reverting is always possible but is not done lightly. Therefore, regressions can be discovered and stay unfixed for much longer, unless someone else is willing to address them.

Another comparison point is a monolithic repository setup hosting a library and all the projects that depend on it in a single company. One of the advantages of this setup is that it makes it easy to evolve an API because the developer updating it will also fix its reverse dependencies [149]. However, this is only possible if all the developers share a common culture, a set of tools and processes, and are ready to tightly synchronize their release process. Our setup makes it possible to assess compatibility breakages, fix reverse dependencies, and yet is compatible with a community having a variety of release cycles, backporting processes, etc.

3.6.6 Open issues, future work

There are three issues with the current setup that I have identified, and that I would like to address in the future: the high number of spurious failures, the long duration of builds, and the complexity and inconsistencies of the overlay preparation workflow.

3.6.6.1 Spurious failures

Spurious failures are failures that are reported in CI but do not reveal any problem in the tested code. They can be caused by various factors such as a broken test-suite, non-deterministic tests that fail from time to time, etc. Frequent spurious failures are a problem because they tend to lower the confidence of contributors and integrators in the results of CI. They can also be a source of inefficiency when a build has to be restarted because of a spurious failure, or when contributors and reviewers lose time trying to figure out the reason of the failure.

In the context of the Coq project, much progress has already been made to address spurious failures due to nondeterministic tests, after I started a project to systematically report and

[37]Example: `https://github.com/scala/scala/pull/7757#issuecomment-470449113`.

address these failures.[38] About 14% of spurious failures are still due to the test-suite, while the rest are mostly due to our compatibility testing infrastructure.

There are two reasons why a tested external project can cause a spurious failure. The first is simply if the author of the project has pushed a bad, untested commit. This is something that can be avoided through systematic CI testing before pushing to the main branches, but it is not clear that we can trust maintainers to always follow this process. The second is due to a synchronization problem, following the integration of a pull request with backward-incompatible overlays.

Typically, a pull request which changes the API is merged in the Coq repository. The author of the pull request had prepared fixes and submitted them as pull requests to the tested plugins, but these pull requests could not be merged until the API change actually occurred. Upon merging the pull request in the Coq repository, the integrator tells the plugin maintainers to merge the corresponding fixes, but this action can be delayed by a few hours to a few days, either because the maintainer is not available immediately, or because they have a policy of only merging pull requests passing CI, and this requires waiting for their testing image of Coq to be updated as well. In the meantime, all builds started in the Coq repository will report (spurious) failures for the corresponding plugins.

My proposed solution to these problems is to stop always testing the latest version of external projects, but instead test the latest known-to-be-working version, and regularly and automatically update this version. Whenever a backward incompatible change is merged, the pinned version is temporarily set to the not-yet-merged commit fixing the incompatibility. Part of the required infrastructure, allowing to pin specific commits of the tested projects, has already been integrated,[39] but I have delayed my work on this issue because of the skepticism expressed by Emilio Gallego Arias, the main author of the overlay system, and despite having gained overall support from the rest of the developers and some external project maintainers.[40] Rolling out this change progressively, starting with just a few plugins, should ensure that it is sufficiently tested, and can gain the support of everyone.

3.6.6.2 Build duration

Counting only the pipelines that are run on GitLab CI, each pull request update spans 58 jobs, including 18 that take longer than 30 minutes to run (and 4 that take longer than 60 minutes), adding to a total sequential time of over 21 hours. Yet, thanks to massive parallelism, these pipelines frequently run in just 105–110 minutes, and failures occurring in the test-suite, or in the plugins, are even reported much sooner to the pull request author.

[38] Cf. `https://github.com/coq/coq/projects/9`.
[39] Cf. `https://github.com/coq/coq/pull/8348`.
[40] Cf. `https://github.com/coq/coq/issues/6724`.

However, this leads to a massive waste of computing resources, that we cannot just ignore blindly in a time when the energy consumption of information and communication technologies is a growing worldwide concern [5].

There are many ways of attacking this problem. One could use techniques such as regression proof selection [48, 218] so that, when the standard library is updated, only the parts of the external projects that depend on the updated code are built again. However, most pull requests in the Coq repository do not just update the standard library, but also the OCaml code. In this case, the compiler must be regenerated, and these techniques no longer apply.

Reusing cached builds of the OCaml code could be used to avoid rebuilding some unaffected OCaml files belonging to Coq or the tested external plugins. However, most of the time in CI is spend building Coq code, not OCaml code.

My proposal to address this problem is to have a more lightweight pipeline run whenever a pull request is opened or updated (just a single build of Coq, the plugins —internal and external—, and the test-suite), and to have the full pipeline only run before merging a pull request. This would reduce the total sequential build time to about 140 minutes, and the parallel build time to about 35 minutes.

However, this change can only be implemented if most issues are already caught by the lightweight pipeline. This is not the case today. As we have seen, more than half of the time a library breaks, the test-suite passes without any failed tests. My plan is to reuse these data points to extract additional test cases that would have failed in these circumstances, and to do so in the most automatic possible manner. To do so, I intend to reuse and possibly improve upon an existing “bug minimizer” for Coq [125].

3.6.6.3 Overlay preparation workflow

Due to the variety of build systems, dependency management methods, platforms hosting the tested external projects, etc., the experience of preparing an overlay is far from uniform, and strongly depends on which projects are concerned. Furthermore, despite our efforts to clearly document the process [50, 102, 327], the experience can be puzzling for a new contributor. To ensure that we do not discourage new contributors to submit certain types of pull requests, we must ensure that the overlay creation process is as straightforward as possible.

Work has already started in this direction. Emilio Gallego Arias proposed an experimental script handling automatically part of the process. Vincent Laporte created Nix files that include the additional dependencies required to build the external projects. The bot could be used to automatically open pull requests on the external projects, whenever overlays are added to a pull request on the Coq repository. This would avoid the need for the contributor to create accounts on a variety of platforms, and to learn how to open pull requests there.

3.7 How to distribute the review workload?

During the initial period of adoption of a pull-based model by the core developers (from October 2016 to March 2018), pull request integration was the task of the release manager alone. This is a frequent setup in open source projects because this gives more control to the release manager (for instance to implement a feature freeze period). However, this is also more demanding, and can significantly delay the integration of pull requests. Reviews were conducted by whoever was available, and there was no clear definition of when a pull request was sufficiently reviewed and could be merged.

Along with our effort to decouple the contributing process from release management, and to streamline the release process (see Chapter 5), we formalized, with Maxime Dénès and the participation of the whole development team, the reviewing and integration process with the goal of distributing the review and integration workload.[41]

Our new process takes advantage of a `CODEOWNERS` file, whose support GitHub had introduced less than a year before [217]. This feature allows to automatically request reviews from maintainers listed in the file, depending on which components are updated by the pull request. Each component (file or directory) is associated to a primary and a secondary maintainer.[42] Their task is to review any incoming pull request updating the component they own (the secondary maintainer handles pull requests from the primary maintainer, and from other people when the principal maintainer is unavailable). Then, one of the code owners that were requested a review should self-assign the pull request and merge it when all the reviewers have approved it.

Furthermore, a merging script [78] is provided, that must be used by the integrators, and which applies a few "sanity checks" such as asking for an explicit bypass when some CI jobs have failed, checking that there are no mistakes regarding which branch the pull request is merged into, checking that there is at least one "kind" label (marking the type of pull request) and no "needs" labels left (which would indicate that something still has to be done before the pull request can be merged, cf. Section 3.5.1), and that a milestone has been set.

This new process has been relatively successful. It has achieved its primary goal which was to relieve the release manager from this task. It has also been an opportunity to request previous contributors who were not part of the core team to participate to code reviews (and to better understand which parts of the code did not have any expert left to maintain them). On the other hand, it creates strong disparities between pull requests, depending on what component they change, and thus who is supposed to integrate them (because not everybody is as responsive). Another unfortunate result is that when the principal maintainer is very reactive but the secondary maintainer is much less reactive, pull requests originating from anyone but the primary maintainer will be much quicker to integrate than pull requests originating from the

[41]Cf. `https://github.com/coq/coq/pull/7014`.

[42]GitHub does not support this distinction, so in practice, only primary maintainers are requested reviews using the code owner system.

primary maintainer.

As a follow-up, and as a way of alleviating some issues that were observed, I started to introduce code owner teams.[43] Code owner teams take advantage of another recently introduced GitHub feature: the ability to request reviews from teams [230]. Code owner teams remove the asymmetry between a primary and a secondary code owner. They also make it possible to have a longer list of code owners for a given component. Finally, they make it easier to add or remove a maintainer, since it only means updating the team membership and does not require an update to the `CODEOWNERS` file. This has further helped involving new people in the reviewing and integration process. It also makes transitions easier: a code owner that is less active can be kept on the team, but the addition of new people ensures that pull requests are not staling.

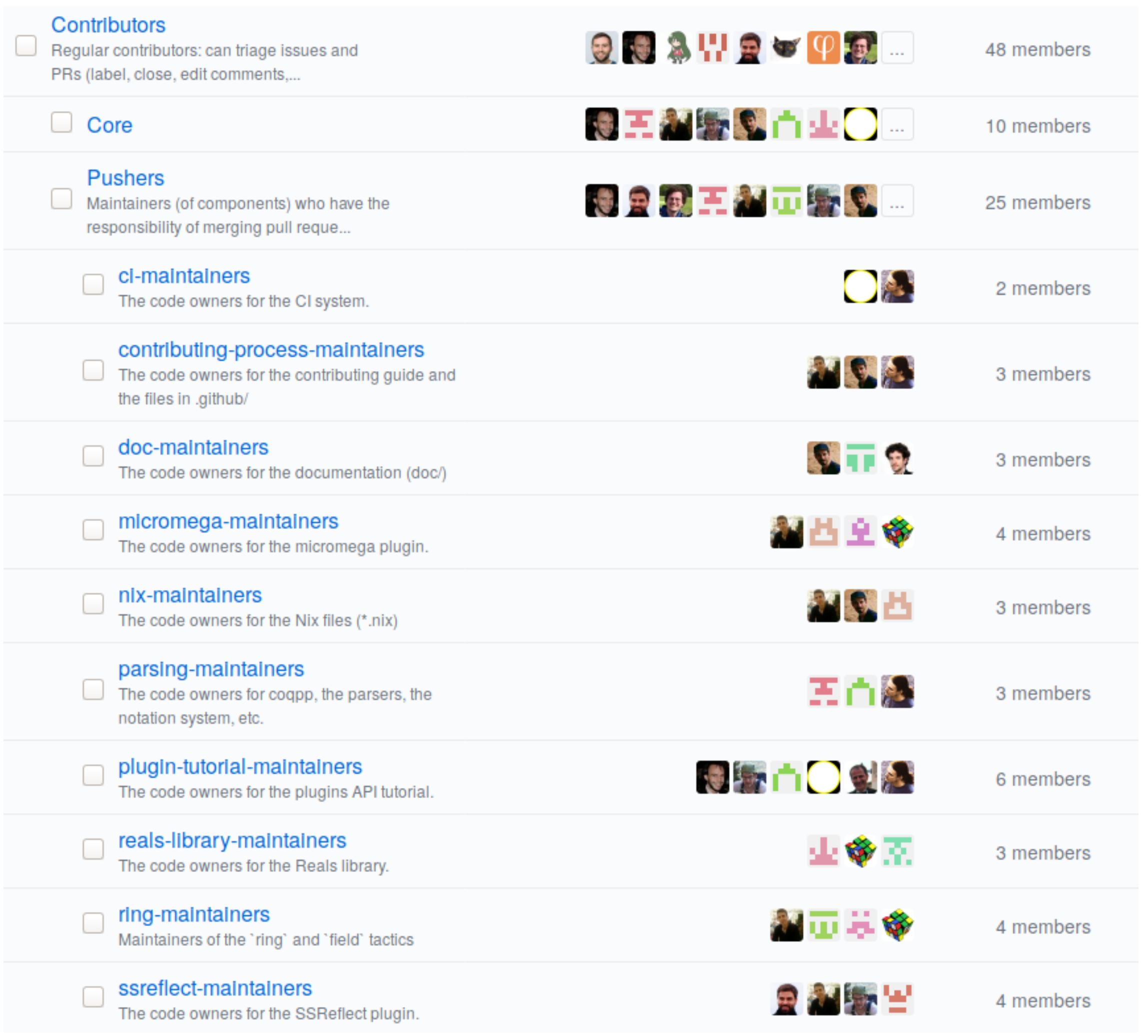
Technically, the introduction of code owners and code owner teams requires first to restrict when main branches can be pushed to, thanks to the GitHub protected branch feature [30, 187, 289]. Access to the master branch is restricted to merging approved pull requests, and the list of people who can push is restricted to a specific “Pushers” team (this is because we give “write access” to a larger “Contributors” team to help with issue triaging, since the move to GitHub issues presented in Chapter 4). Access to the release branches is also restricted to approved pull requests, and only the users with “admin access” (i.e. the members of the “Core” team) can push to them (in practice, this stays the role of the release manager, cf. Chapter 5). Code owner teams are created as sub-teams of the “Pushers” team, which means that their members are automatically members of the “Pushers” team as well. Figure 3.4 shows a screenshot of the organization’s internal team page where we can see the various teams and their membership. The “Pushers” team is now 25 members strong, which is to compare to a core team of only 10 people.

3.7.1 Open issues, future work

While the merging script is an efficient way of reminding a diversified team of integrators about some criteria that must be satisfied for a pull request to be merged, it also creates a barrier of entry. Indeed, this script requires some dependencies to be installed, a certain setup of the integrator’s local clone of the Coq repository, and that the integrator has a GPG key associated to their git identity. A way to lower this barrier of entry would be to avoid the integrators to use this script, by allowing them to merge a pull request by issuing a command to the bot. In fact, I should be able to add this feature soon, because most of the required infrastructure, in particular to check that the person issuing the command belongs to the appropriate team, is already implemented.

Despite the progress that has been made regarding the pull request integration process, it is still not satisfying on several aspects, since it receives regular complaints coming from a few

[43]See `https://github.com/coq/coq/pull/8023`, `https://github.com/coq/coq/pull/8250`, `https://github.com/coq/coq/pull/8430`, `https://github.com/coq/coq/pull/8924`, and `https://github.com/coq/coq/pull/9827`.



Team	Description	Members
Contributors	Regular contributors: can triage issues and PRs (label, close, edit comments,...	48 members
Core		10 members
Pushers	Maintainers (of components) who have the responsibility of merging pull reque...	25 members
ci-maintainers	The code owners for the CI system.	2 members
contributing-process-maintainers	The code owners for the contributing guide and the files in .github/	3 members
doc-maintainers	The code owners for the documentation (doc/)	3 members
micromega-maintainers	The code owners for the micromega plugin.	4 members
nix-maintainers	The code owners for the Nix files (*.nix)	3 members
parsing-maintainers	The code owners for coqpp, the parsers, the notation system, etc.	3 members
plugin-tutorial-maintainers	The code owners for the plugins API tutorial.	6 members
reals-library-maintainers	The code owners for the Reals library.	3 members
ring-maintainers	Maintainers of the `ring` and `field` tactics	4 members
ssreflect-maintainers	The code owners for the SSReflect plugin.	4 members

Figure 3.4: The list of teams with access to the Coq repository as of September 11th, 2019. The “Contributors” team members have write access without the ability to push, to be able to participate to issue and pull request triaging. The “Core” team reflects who are the administrators of the Coq organization. The “Pushers” team members have the permission to merge approved pull requests to the master branch. The various sub-teams of the “Pushers” team are the code owners of various components.

developers. In particular, they complain that pull request integration is too slow, that reviewers do not always provide good feedback, and sometimes request changes on unsound bases, that there is no process to make discussions converge, and finally that decisions to integrate a pull request can sometimes happen too suddenly, and unpredictably, without time for concerned developers to comment, resulting in an overall unfair process.

Some of these impressions might correspond to actual problems worth addressing, and even when they are just impressions, it is important to be able to understand why developers feel that way. A first step will therefore be to interview developers to list their complaints, and to gather data to try to back up these complaints. In particular, it is clear that pull request integration efficiency strongly varies depending on the affected components, and therefore may lead different developers to have different perceptions.

Once issues are more clearly identified, it should be possible to decide what is the best solution to address them: whether it is clearer processes, better tooling, more community involvement, or only conscious efforts from concerned developers.

3.8 Conclusion

In this chapter, I have presented the challenges associated with the adoption of pull requests in the development of Coq. The experience report from this project confirms what was observed by Dias *et al.* [79]: that moving to GitHub alone is not sufficient to attract more contributions. In fact, for the first few years, mirroring the Coq project on GitHub, and even switching to git, did not impact the development model, or the number of contributors.

However, when the development team decided to take specific steps to attract more contributors, such as organizing workshops, opening a chatroom, improving the developers' documentation, and in particular providing clear contribution guidelines, this was followed by a strong increase in the participation of external contributors. Some new contributors even participate now to the integration of pull requests.

It is worth noting that this contrasts with the result that will be presented in Chapter 4: that a simple switch of bug tracker to GitHub had strong effects on both the level of bug tracking activity, and the participation of external contributors.

Besides the impact on the number of contributors, the switch to a pull-based development model was also an opportunity to improve the quality of integrated changes. I have shown how the introduction of pull request templates with a checklist triggered an increase in the proportion of pull requests including documentation, a changelog entry, and tests. In particular, the proportion of feature pull requests including a changelog entry increased dramatically.

I have also presented our innovative model for *a priori* compatibility testing, and how it has helped produce more stable releases, where compatibility breaking changes are clearly acknowledged, and migration paths are documented.

CHAPTER 4

IMPACT OF SWITCHING BUG TRACKERS

4.1 Introduction

Bug reporting is an essential part of software development. In the context of an open source project, the bug reporting and fixing process is generally done on a public bug tracking platform, and, in the absence of paid testers, it depends a lot on the goodwill of independent users. Therefore, it involves a strong social component.

However, having an open bug tracking system is not enough to attract participants. In addition to the software having enough users, the process of opening an issue (be it bug report or feature request) must be easy and appealing. Therefore, creating a favorable bug tracking environment may lead to an increase in bug tracking activity (from both developers and independent users), which may result in an increase in software quality. First, the participation of more users is helpful to find a larger proportion of bugs [239, 286]. Second, assuming that developers are equipped to cope with the increased number of incoming reports [12, 71], even duplicate reports are not necessarily harmful [28]. Third, more activity on the bug tracker can also mean users are helping to reproduce bugs, produce traces, etc. and thus are working with the developers to get the bugs fixed [40]. More generally, opening issues and discussing existing ones has been shown to be an important step on the path to becoming an active contributor of an open source project [150, 202, 310].

Despite the importance of the bug tracking environment that I have just highlighted, the impact of a change in this environment has rarely been studied, whether it is the bug tracking platform in full, a feature, or a policy [4]. Furthermore, since the choice of bug tracker is not independent of the characteristics of the project, a comparison across projects using different bug trackers would not be sufficient to draw causal conclusions.

In this chapter, I present and study the case of the switch of Coq's bug tracker, starting with the context of the switch and the technical challenges associated to it, and followed by an empirical evaluation of its impact.

My contributions are practical, empirical, and methodological. First, I have improved an existing bug tracker migration tool to handle thousands of issues while preserving meta-data, and managed the switch of Coq's bug tracker from Bugzilla to GitHub, including the migration of preexisting issues. Second, with Annalí Casanueva Artís, we have analyzed the causal impact of this switch on the bug tracking activity through mining repository data, and interpreted the results through qualitative assessment based on interviews with developers. Our results show that the switch incurred an increase in bug tracking activity of core developers, and an increased number of community participants taking part to discussions. Third, we demonstrate in much detail the application of the RDD methodology presented in Section 2.3.2.

This chapter is based on a paper that was accepted at the 2019 International Conference on Software Maintenance and Engineering [324].

4.2 Related work

While there is a very large literature on many aspects of bug reporting (see Strate and Laplante [269], and Zhang *et al.* [312] for literature reviews), there is a lack of literature measuring the influence of the bug reporting environment on the bug reporting activity, or on any other aspect of software development. To my knowledge, there is no previous work measuring the impact of a change in bug reporting environment. More generally, there is little literature that compares bug trackers.

4.2.1 Comparing and proposing features of bug trackers

A simple, but quite limited way to compare bug trackers is to look at the features that each of them proposes. Karre *et al.* [159] compared 31 bug trackers feature-wise, and gathered these tools in four clusters. Bugzilla belongs to the cluster of bug trackers with many features (together with RedMine and Mantis), while GitHub belongs to a cluster of bug trackers attached to source code management systems (together with Savannah and BitBucket). Abaee and Guru [1] listed the features of four commercial bug trackers, and proposed their own bug tracker with some new features, but with no evaluation.

Many papers propose new features to add to bug trackers, but they rarely evaluate the impact of adding such features, even when a prototype was presented. Baysal *et al.* [23] identified the need for personalized issue tracking systems after interviewing twenty Mozilla developers. They proposed a Bugzilla extension to address this need, and gave a, mostly qualitative, assessment by interviewing developers using their tool [24]. Bortis [36] developed a bug triaging tool and evaluated how users interacted with it. However, there was no evaluation of its impact in the context of an actual software project. Just *et al.* [154] surveyed developers from three large open source projects (Apache, Eclipse, and Mozilla –all three projects use Bugzilla) to identify missing features and provided recommendation to design better bug tracking systems.

Our evaluation goes beyond a descriptive or normative perspective and quantitatively measures the causal impact of the change of the bug tracking platform (a crucial part of the bug tracking environment) on various aspects of bug tracking activity, and gives qualitative insights for its interpretation.

4.2.2 Analysis of projects' bug tracking data

Another way of comparing bug tracking systems could have been to conduct large-scale studies of many software projects using various bug trackers, and derive some differences associated with the use of the various systems. However, there are no such comparative studies in the literature.

Sowe *et al.* [259] noted that most preceding literature had only been comparing few projects at a time, usually using a single bug tracking system. In their study, they addressed this in part by studying hundreds of projects, but they did not mention which bug trackers the projects they compared used.

Bissyandé *et al.* [29] published the same year a larger scale investigation using ten thousand projects' bug tracking data, analyzing such things as the correlation between a project's success and its bug tracking activity. All the projects studied were using the same bug tracker (GitHub's).

Francalanci and Merlo [96] analyzed the bug fixing process by studying closed bugs from nine open source projects (four using JIRA, four using SourceForge's bug tracker, and one using another bug tracker). They did not, however, try to correlate the bug tracking activity with the bug tracking system.

4.2.3 Switching support channels

Squire [263] measured the impact of switching developer support channels from mailing lists and self-hosted forums to Stack Overflow, and compared it with the expectations of projects having decided on such a move, whereas Vasilescu *et al.* [288] compared the behavior of the same users during the same period on the R mailing list and on Stack Overflow.

An advantage with support channels is that they can be diversified, and it is possible to experiment with a new platform without abandoning the previous one (for instance, Coq's support channels currently include a users' mailing list, Stack Overflow, and a Discourse forum). On the contrary, bug tracking data is preferably located in a single, central location, and thus a decision to switch platforms is not easily reverted (especially when bug tracking data is migrated). Therefore, it is all the more important to provide empirical evidence of the consequences.

4.3 Switching Coq's bug tracker

4.3.1 Context

From 2007 to 2017, the Coq bug tracker platform was a self-hosted Bugzilla [21] instance (before this, from 2001 to 2007 it was using JitterBug [281], a now discontinued bug tracking system, and before 2001 a simple mailing list). In 2017, in a context where all code changes were conducted through GitHub pull requests (cf. Chapter 3), the question of switching to GitHub's integrated bug tracker arose. After I conducted some preliminary testing that showed that migrating all preexisting issues was feasible, the Coq development team approved the switch to GitHub issues on October 4th, 2017.

According to the developers we interviewed, the motivations for switching were:

- Consolidating the development tools on a single, integrated, platform: browsing between code, pull requests and issues is easier without having to switch websites. Furthermore, it means a common notification, mention, and assignment system.
- GitHub's support for formatting and editing comments, cross-referencing and auto-closing issues, and more generally a more pleasant tool to use. Many developers perceived Bugzilla as unpleasant, old-fashioned, and very slow. It was a complex tool with many underused advanced features.
- No need to administrate (maintain and update) the bug tracker. Almost half of the developers complained about the previous bug tracker failing often.
- Easier to get started for newcomers (especially as many may already know GitHub and have a GitHub account).

Again, according to the developers we interviewed, the principal risks associated with the switch of bug tracker were:

- Losing control of a critical tool to a private company and its future decisions. In particular, GitHub is not an open source platform (so cannot be cloned elsewhere) and backing up data from GitHub becomes even more important, but not trivial to do.
- Losing or corrupting bug tracking data during the transition from Bugzilla to GitHub. During the migration, some meta-data had to be converted to text, and external links and documentation about the bug tracker could have become stale or inaccurate.
- A few developers additionally mentioned risks of losing useful features from Bugzilla, or discouraging some developers for whom it would be hard to adapt to the new tool.

4.3.2 Migration

As explained in Section 4.3.1, one of the main risks (and deterrents) related to a switch of bug tracker was the possible loss or corruption of bug tracking and associated data. In particular, the Coq development team wanted preexisting issues to be migrated to the new bug tracker, while keeping the issue numbers as much as possible: indeed, these numbers are important because they are mentioned in many places (changelog, test-suite, other issues and pull requests, commits, external discussion platforms, etc.).

I reused and adapted a tool [26] which is designed to import Bugzilla issues (extracted as an XML dump) to GitHub using its REST API [112]. The issues are imported in an order designed to preserve numbers whenever possible. Issues whose number is unavailable (e.g. because the number is already taken by a GitHub pull request) are postponed and renumbered.

I implemented several changes to make the tool better fit the needs of the Coq project (the main improvements having now been integrated upstream[1]):

Allowing non-consecutive numbers The imported set of issues had some holes in the numbering because some issues had been deleted. In particular, the previous migration from JitterBug to Bugzilla had not preserved issues that were already closed at the time, so the beginning of the set had many such holes. I added support for non-consecutive issue numbers by using postponed issues to fill the holes (in practice, there were sufficiently many postponed issues to fill all the holes). The current version of the tool can create dummy issues when the set of postponed issues is empty.

Saving a correspondence table for renumbered issues This was suggested when I requested a review of the adapted migration script from other developers,[2] and turned out to be particularly useful, as this was used later on to redirect the old Bugzilla URLs to the new GitHub ones, and to rename some test files corresponding to renumbered bugs.[3]

Using GitHub's issue import API and overcoming GitHub's rate limits Creating a new issue or a new comment through the normal GitHub REST API triggers notifications (for people who are watching the repository or are mentioned in the issue thread). Therefore, GitHub chooses to impose a strict rate limit on these actions, which prevented using this tool for importing more than a few hundred issues. Fortunately, GitHub provides a (beta) issue import API [139] which, in addition to not triggering notifications, also allows importing one issue, its comments, and meta information such as closed status and assignee in a single request (thus reducing the duration to import 4900 issues to just a few hours). Furthermore, using this API allows to keep the dates of

[1]Thanks to the initial help of Martin Michlmayr. See `https://github.com/berestovskyy/bugzilla2github/pull/3` and `https://github.com/berestovskyy/bugzilla2github/pull/4`.

[2]See `https://github.com/coq/coq/pull/1148`.

[3]See: `https://github.com/coq/coq/issues/6001`.

imported issues and comments, which was very useful during the empirical evaluation of the switch.

Once the switch was approved by the development team on October 4$^{\text{th}}$, 2017, it had to happen as soon as possible because every new pull request before the migration added to the number of issues that would need to be renumbered. It was conducted on October 18$^{\text{th}}$, the day after the 8.7.0 release (and not before to avoid disturbing the release process). Only 502 out of 4900 issues (whose numbers were below 1154) had to be renumbered. This number is quite low because a lot of the most ancient issues had been deleted (during the previous bug tracker migration). Due to the rate of pull request creation, if the switch had been delayed by a single year, the number of renumbered bugs would have been closer to 3000.

4.4 Empirical validation

In this section, I present the empirical evaluation of the impact of the bug tracker switch that we have conducted with Annalí Casanueva Artís. We conducted a quantitative evaluation by applying the RDD methodology presented in Section 2.3.2 to mined software repository data. We completed this analysis by a qualitative evaluation based on interviews with Coq developers, which allows us to interpret the quantitative results with more confidence, and gives additional insights on the impact of the bug tracker switch.

4.4.1 Research questions

Our research questions cover the spectrum of the possible impact such a switch could have had on the bug tracking activity:

Impact on level of bug tracking activity (RQ1) Did the switch to the GitHub bug tracker, given the various expected benefits that such a switch would bring, impact the level of activity on the bug tracker? If so, who are the participants whose level of activity was impacted, and how can we explain such changes?

Impact on quality of bug tracking activity (RQ2) Did the switch to GitHub impact the quality of opened issues, and did it impact the quality of interactions between developers, and between developers and non-developers?

Impact on the audience of the bug tracker (RQ3) Did the switch to GitHub help onboard more new reporters and commentators? Did it increase the number of distinct non-developers that regularly take part in the bug tracking activity?

4.4.2 Data

4.4.2.1 Extraction

All the data for this study was extracted on March 31st, 2019 using the GitHub GraphQL API [117]. Using this API allows us to do large requests (100 nodes in a single request) with just the information we need (thus both reducing the bandwidth usage and speeding up the extraction process). In the companion Jupyter notebook [323], we provide the code to request this data from GitHub, to load the CSV files, to run the pre-processing steps and the analyses, including the robustness checks presented in Section 4.4.6.1.

As was previously mentioned, the migration conserved the dates of issues and comments, which allows us to obtain them transparently for issues before and after the switch. On the other hand, author information for migrated issues and comments had to be encoded in the text, and is thus extracted from there. Finally, we do not consider any data from before 2008, because this data comes from a previous migration, and was not properly saved.

4.4.2.2 Pre-processing

Excluding specific reporters We have one specific reporter who is alone responsible for almost a quarter of all issues. To avoid having the behavior of a single individual strongly impact the overall statistics, we exclude his comments, issues, and the comments they received from our analysis. Similarly, we also exclude my own activity from the dataset.

Merging duplicate accounts A significant number of Bugzilla accounts were duplicates, i.e. they belonged to users who had created several accounts. This typically happened when people used different e-mail addresses (generally belonging to their successive employers / institutions). These duplicate accounts were merged in three (manual) steps:

- First, during the migration process, I created a correspondence table that mapped 217 (out of 686) Bugzilla accounts to 175 GitHub accounts. Priority was given to finding GitHub accounts for users who still had opened issues.
- Second, when a Bugzilla user who had not been mapped to a GitHub account ended up being active on GitHub (for instance, because they created a GitHub account after the switch), then I edited migrated issues to use their GitHub username, as if they had been listed in the correspondence table.
- Third, after extracting the data from GitHub, I further identified duplicate Bugzilla accounts (that had not been merged because they had not been mapped to GitHub accounts).

The size of the dataset allowed this merging step to be a manual process. Larger datasets would have made the use of identity merging heuristics necessary. Usually, the duplicate accounts were easy to identify because they provided the full name explicitly or the e-mail address contained it. In the few cases where there remained ambiguity, I searched for more information about the users on the internet (benefiting from the fact that a lot of them were from academia and used their institution e-mail addresses).

We did not merge any GitHub accounts. It is indeed much less likely that someone forgets about their existing GitHub account and creates a new one. In practice, I am aware of a single reporter having used several GitHub accounts (a personal and a professional one), but this dates from after our data collection.

Even if it is possible that a few more duplicate accounts have been missed, we strongly believe that it would not change anything to our results:

- It could have had an effect on our new reporter or commentator analysis, but we do not have any statistically significant results for these.
- It is very unlikely that it could affect our analysis of distinct weekly reporters or commentators, because that would mean that someone used some duplicate accounts (that were missed during the merging phase) during the same week.

Removal of migration artifact comments The migration tool created artifact comments, which are easily identified because they are the comments that were posted at the exact same date and time as the corresponding issue. We exclude them from our analysis.

4.4.2.3 Variable definition

We measure different indicators of bug tracker activity: the numbers of issues per day; the number of distinct reporters in a week; the number of new reporters (who had never opened an issue before) per day; the number of comments per day; the number of distinct commentators in a week; and the number of new commentators per day. The number of distinct reporters and commentators is intended to allow us to distinguish between having a few prolific contributors and a large base of casual contributors. We use an interval of a week instead of a day for measuring the number of distinct issue authors and commentators because, at the scale of a day, there is less opportunity for repeated contributions by the same contributor, but longer scales would compromise the estimation of effects by removing too much data.

In the figures of Section 4.4.3, each point represents an average on a four-week period. This is to reduce variability and allow an easier visual analysis. In the figures of Section 4.4.4, each point represents an average on a two-week period.

We analyze heterogeneous effects by distinguishing between developers and non-developers. We define “developers” as the persons who have contributed more than 100 commits since 2008. We identified 18 developers. These developers are responsible for 91.5% of all commits since 2008 (this is consistent with standard results on the proportion of commits by the “core team” in open source software [248]). Out of the 18 developers, 11 were active in the two months preceding the bug tracker switch (committed at least once). Once we exclude the two reporters mentioned in Section 4.4.2.2, there remain 9 developers that were active at the time of the migration and that were included in our analysis. We later interviewed all of them. This seemingly arbitrary criterion happens to correctly recover what Coq developers viewed as the "core team" during the period around the bug tracker switch.

4.4.3 Descriptive statistics

Figures 4.1, 4.2, 4.3, and 4.4 show the evolution of our four main outcomes from January 2016 to March 2019. We choose to present the data starting in 2016 because, in January 2016, Coq 8.5 was released, and this marks the adoption of a more rapid release cycle (see Section 5.2). This change in the development process could have impacted the activity on the bug tracker and made it not comparable.

In Figure 4.1 and 4.2, each point represents the 4-week average of the number of issues and number of comments per day respectively for both developers and non-developers. In Figure 4.3 and 4.4, each point represents the 4-week average of the number of distinct reporters and distinct commentators per week. In all four figures, the vertical red line shows the date of the bug tracker switch; the vertical black lines represent the date of major releases and the discontinuous horizontal lines represent the mean before and after the switch respectively.

Considering the whole period (2016–2019), less issues are reported by developers than by non-developers. In a four-day period, on average, two issues are opened by developers and five by non-developers. On the other hand, developers post more comments than non-developers. On an average day, developers post around five comments and non-developers two. This is not surprising because some bugs found by developers are resolved directly without opening an issue, and because issue reports are generally answered by developers. There are more distinct non-developers than developers who open issues in an average week (around six distinct non-developers and two distinct developers). On average on the whole period, there are slightly more distinct non-developers (per week) that comment than developers. Interestingly, this is a case where the switch of bug tracker marks a change in the ranking: before the switch there were more distinct developers that commented each week, and afterwards, there were more distinct non-developers.

For all outcomes, the mean before the switch is statistically significantly lower than the mean after the switch both for developers and non-developers (mean difference T-tests, $p < 0.001$). For some outcomes, the difference in activity patterns before and after the switch is clear to the naked eye. In Figure 4.1, we see an increase in the number of issues reported by developers. In Figure 4.2, we see an increase in the number of comments by developers with many data points after the switch that are well above the highest point before the switch. Finally, in Figure 4.4, we see a clear increase in the number of distinct non-developer commentators with almost all

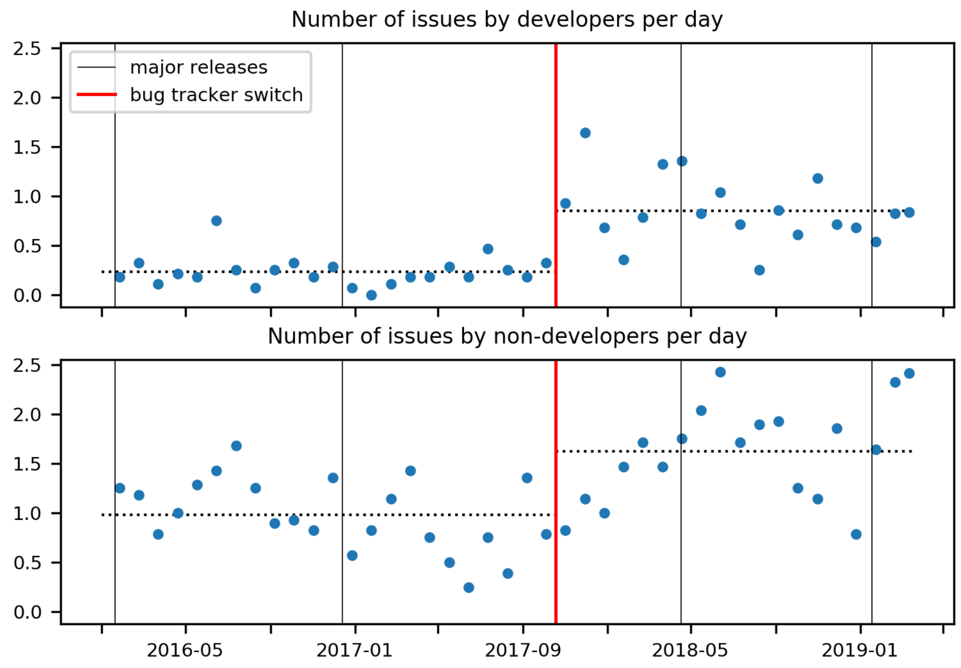


Figure 4.1: Number of issues per day (averaged by 4-week periods) with release dates (since 2016).

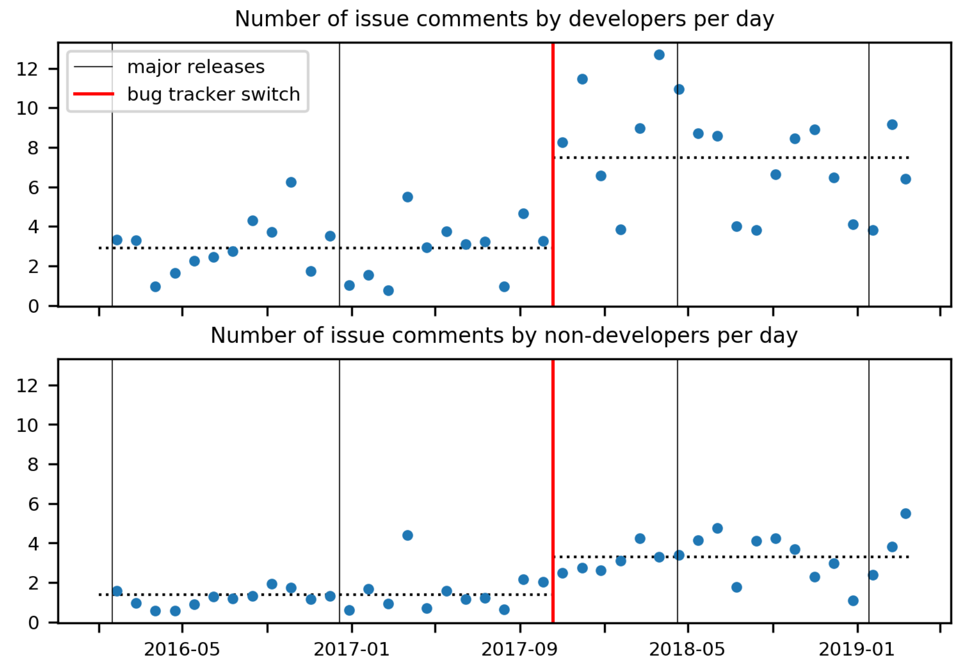


Figure 4.2: Number of comments per day (averaged by 4-week periods) with release dates (since 2016).

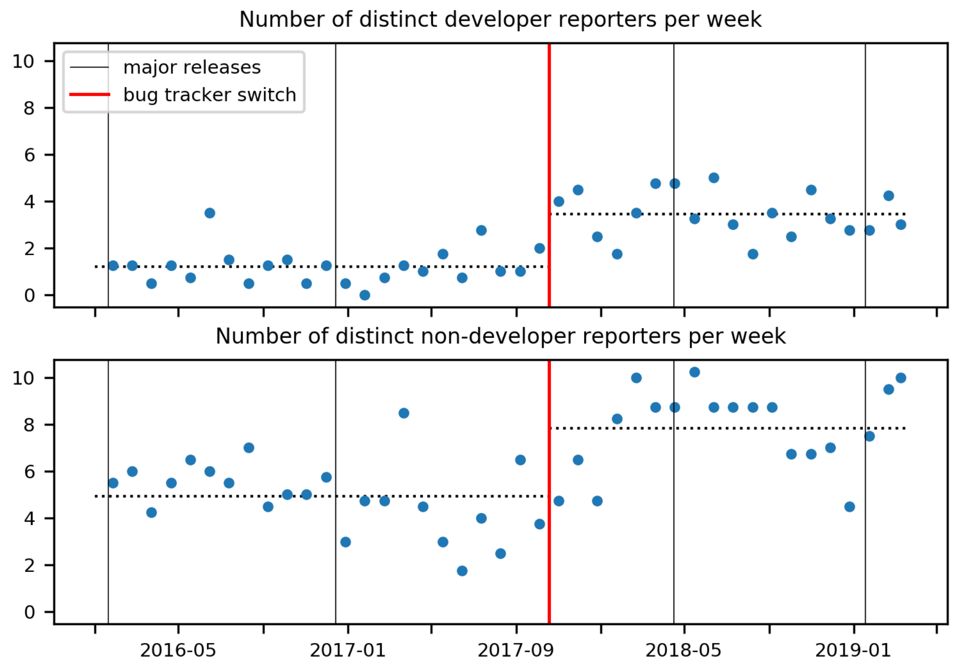


Figure 4.3: Number of weekly distinct reporters (averaged by 4-week periods) with release dates (since 2016).

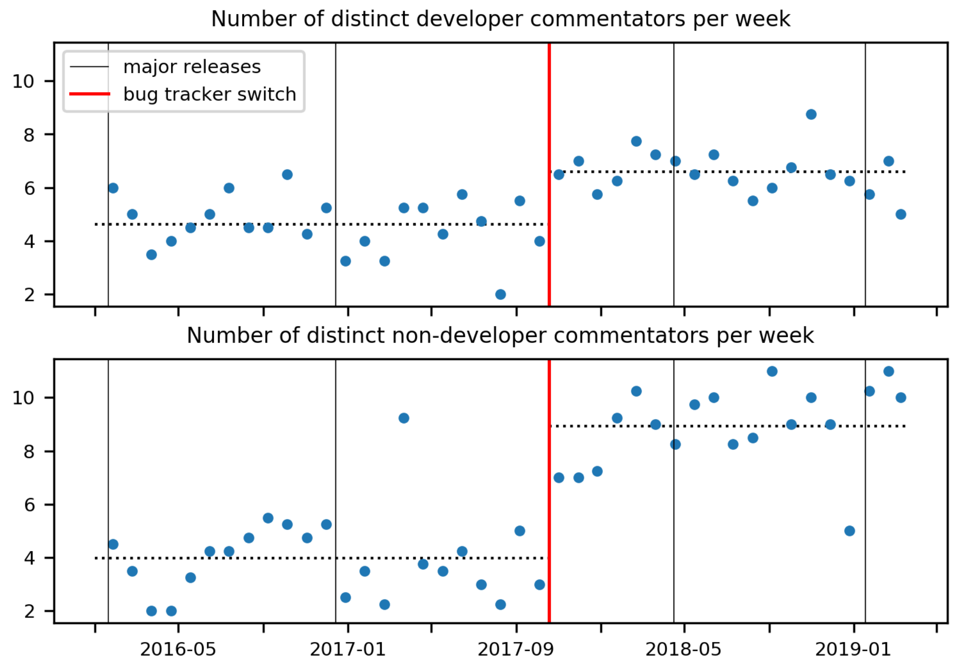


Figure 4.4: Number of weekly distinct commentators (averaged by 4-week periods) with release dates (since 2016).

	Total	Developers	Non-developers
After switch$_p$	0.629	**0.73****	-0.101
	(0.354)	(0.239)	(0.244)
Relative date$_p$	0.00114	-0.000383	0.00152
×*After switch*$_p$	(0.00341)	(0.00227)	(0.00245)
Relative date$_p$	0.00305	-0.00013	**0.00318***
	(0.00182)	(0.000816)	(0.00157)
Constant	**1.23*****	**0.269*****	**0.96*****
	(0.21)	(0.0809)	(0.181)
Observation number	350	350	350

Table 4.1: Estimated impact on the number of issues. Statistically significant results are in boldface (* means $p < 0.05$, ** means $p < 0.01$, *** means $p < 0.001$). Standard error is in parentheses.

points after the switch above almost all points before the switch. Some other differences may be appreciated but are less clear to the naked eye than those presented.

4.4.4 Causal analysis of the impact of the switch

We exploit the fact that the switch from Bugzilla to GitHub can be seen as a clear cutoff in time to use the RDD methodology presented in Section 2.3.2 to estimate the effects of the switch on our outcome variables, to determine if the estimated effects are statistically significant, and to interpret them causally.

Our main specification takes a conservative approach with a relatively small bandwidth of 175 days before and after the switch to minimize possible bias, and a simple linear model to avoid overfitting. However, as a robustness check, we also estimate a quadratic model on a larger time frame (511 days on each side, cf. Section 4.4.6.1).

We estimate our RDDs for two different sub-samples: the developers and the non-developers.

4.4.4.1 Impact on the number of issues

Table 4.1 presents the estimated impact of the switch on the number of issues. Each column shows the estimates for a different sub-sample (all reporters, developers and non-developers). Estimates that are not statistically significant cannot be interpreted as an absence of effect but indicate that if such effect exists, we are unable to discern it with our conservative approach. Figure 4.5 shows the number of issues before and after the switch and the fitting lines and confidence intervals corresponding to the regression results.

For developers, we see a statistically significant positive jump in the rate of issue creation just after the switch. The switch is estimated to have increased the daily number of issues by 0.7 (representing an increase of around 270%). That is, on average, we observe around one issue

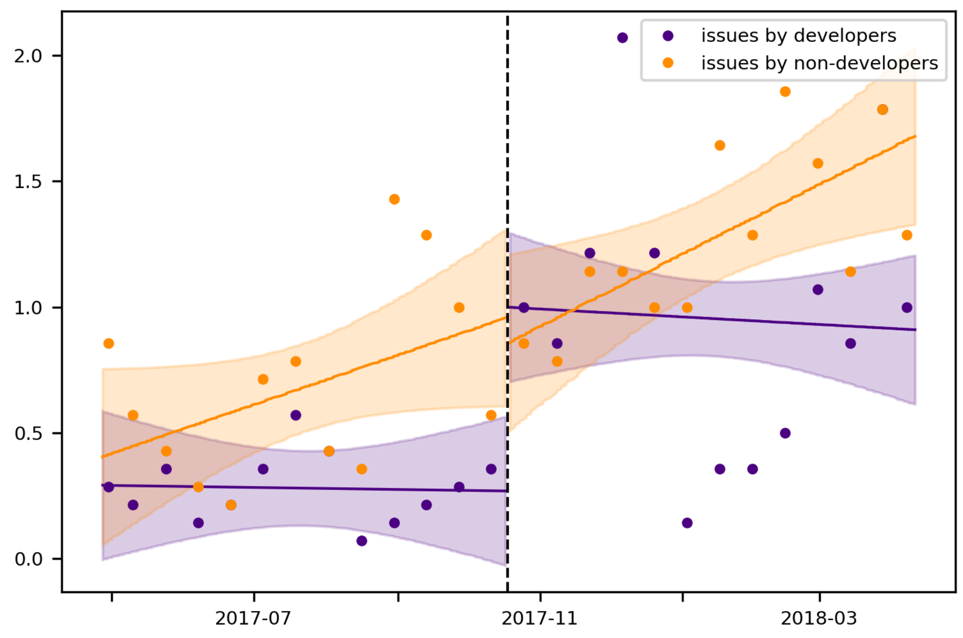


Figure 4.5: Number of issues per day before and after the switch (with fitting lines and confidence intervals from the regression results, and points corresponding to average values over two-week periods).

opened by developers every day after the switch, while, before the switch, developers opened an issue every four days. There is no statistically significant effect on the number of issues by non-developers.

4.4.4.2 Impact on the number of comments

Table 4.2 and Figure 4.6 show the estimated impact of the bug tracker switch on the number of comments. We see a statistically significant positive jump. The number of comments just after the switch more than doubles (from around 5 comments per day before the switch to around 10 after). This appears to be due, for the most part, to comments by developers who increased their total average number of comments per day from around 3 to around 8.

4.4.4.3 Impact on the number of distinct reporters

Table 4.3 and Figure 4.7 show the estimated impact of the switch on the number of distinct issue reporters each week. These results show that the switch had also a positive effect on the number of distinct developer-reporters in a given week. The number of distinct developers that opened an issue in a given week increased, on average, by around 130% (from 1.5 developer-reporters each week before the switch to 3.42 after).

	Total	Developers	Non-developers
After switch$_p$	**5.39***	**4.63***	0.76
	(2.29)	(1.83)	(0.729)
Relative date$_p$ ×*After switch*$_p$	0.0156	0.0139	0.00167
	(0.0217)	(0.018)	(0.00655)
Relative date$_p$	0.00235	-0.00156	0.00391
	(0.00909)	(0.00685)	(0.00405)
Constant	**4.82*****	**3.03*****	**1.79*****
	(1.04)	(0.708)	(0.474)
Observation number	350	350	350

Table 4.2: Estimated impact on the number of comments. Statistically significant results are in boldface (* means $p < 0.05$, ** means $p < 0.01$, *** means $p < 0.001$). Standard error is in parentheses.

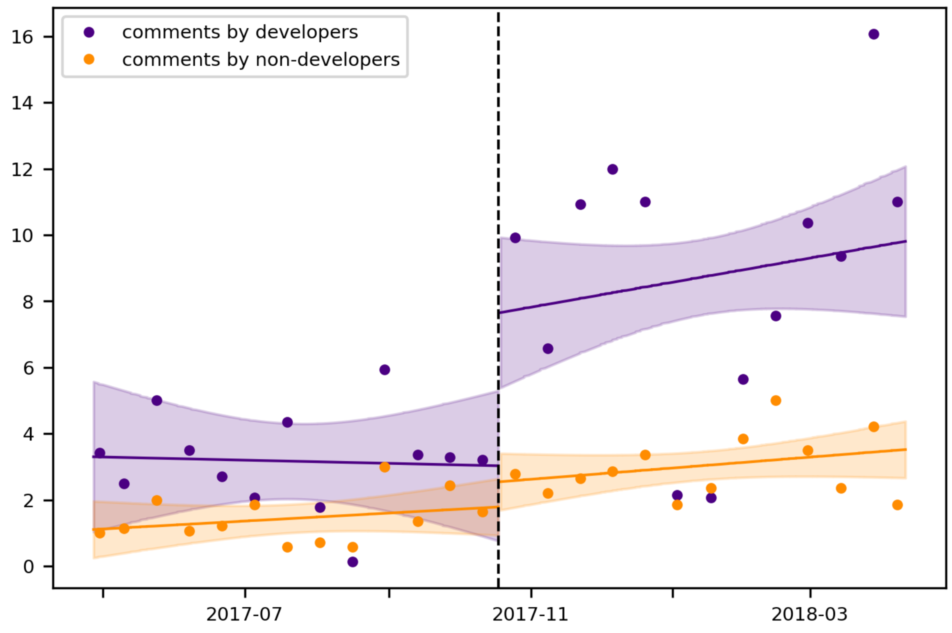


Figure 4.6: Number of comments per day before and after the switch (with fitting lines and confidence intervals from the regression results, and points corresponding to average values over two-week periods).

	Total	Developers	Non-developers
After switch$_p$	2.17 (1.68)	**1.92*** (0.977)	0.245 (1.27)
Relative date$_p$ ×*After switch*$_p$	0.146 (0.108)	0.0162 (0.0648)	0.13 (0.0862)
Relative date$_p$	0.0608 (0.0903)	-0.00462 (0.0378)	0.0654 (0.0675)
Constant	**6.03***** (1.47)	**1.5**** (0.579)	**4.53***** (1.11)
Observation number	50	50	50

Table 4.3: Estimated impact on the number of weekly distinct reporters. Statistically significant results are in boldface (* means $p < 0.05$, ** means $p < 0.01$, *** means $p < 0.001$). Standard error is in parentheses.

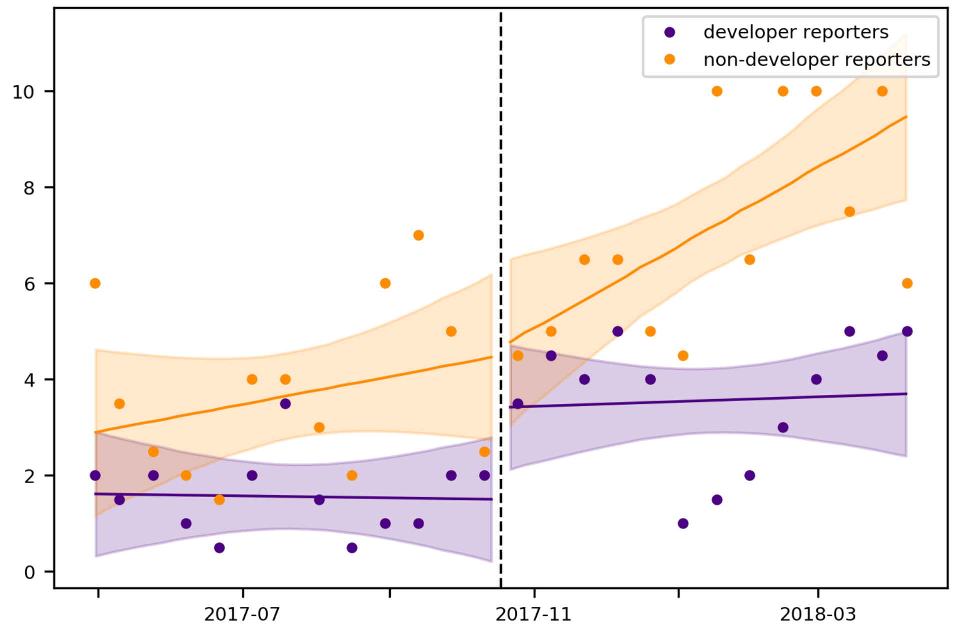


Figure 4.7: Number of weekly distinct reporters before and after the switch (with fitting lines and confidence intervals from the regression results, and points corresponding to average values over two-week periods).

	Total	Developers	Non-developers
After switch$_p$	**6.19***** (1.81)	**2.53*** (1.05)	**3.66**** (1.17)
Relative date$_p$ ×*After switch*$_p$	0.205 (0.118)	0.0769 (0.0641)	0.128 (0.0924)
Relative date$_p$	-0.0692 (0.0833)	-0.0469 (0.0589)	-0.0223 (0.0463)
Constant	**7.06***** (1.54)	**3.83***** (0.984)	**3.23***** (0.861)
Observation number	50	50	50

Table 4.4: Estimated impact on the number of weekly distinct commentators. Statistically significant results are in boldface (* means $p < 0.05$, ** means $p < 0.01$, *** means $p < 0.001$). Standard error is in parentheses.

4.4.4.4 Impact on the number of distinct commentators

Table 4.4 and Figure 4.8 show the estimated impact of the switch on the number of distinct commentators each week. We observe a statistically significant jump in the number of both developer and non-developer commentators.

In general terms, the number of distinct commentators in a given week changed from an average of 7 to an average of 13 (almost doubling the commentators). This increase is due in larger part to the number of non-developers commentators, which more than doubles (from around 3 to around 7 on an average week), while, among developers, the number of commentators increases by 66% (from around 4 to around 6).

4.4.4.5 Impact on the number of new reporters and commentators

The companion Jupyter notebook [323] additionally includes an analysis on new reporters and new commentators. The estimated coefficients also show a positive jump, but they are not statistically significant.

4.4.5 Interviews of developers, discussion of the research questions

In order to obtain qualitative insights into the effects of the switch that will help us interpret the results of the causal analysis, we conducted semi-structured interviews of all developers of Coq that were active at the time of the switch (as defined in Section 4.4.2.3) and that we included in our quantitative analysis. A total of 9 interviews were conducted between March 19th and April 4th, 2019. Both Annalí Casanueva Artís and myself were present during the interviews and all the interviews were recorded. The answers were coded by both interviewers in an attempt to minimize errors. The interview outline with questions asked to all developers, and the encoding of the answers can be found in the supplementary material of the ICSME paper [325].

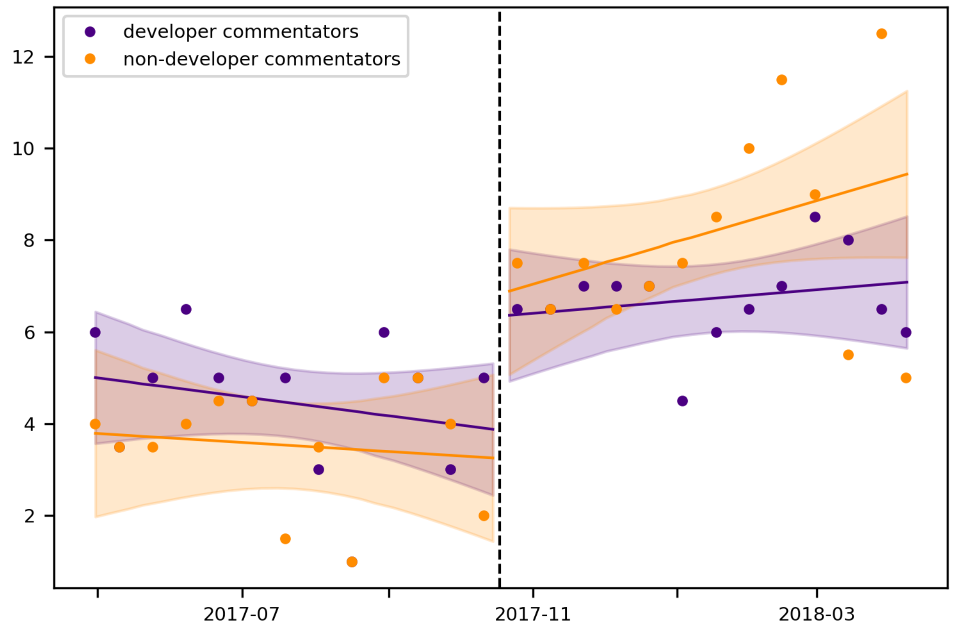


Figure 4.8: Number of weekly distinct commentators before and after the switch (with fitting lines and confidence intervals from the regression results, and points corresponding to average values over two-week periods).

4.4.5.1 RQ1: Impact on level of bug tracking activity

In Section 4.4.3, we observed an increase in the mean number of issues and comments after the switch, both for developers and non-developers. However, as shown in Section 4.4.4, we are able to find a causal effect of the switch on these two variables only for developers. In particular, the estimated impact of the switch was to more than quadruple the number of opened issues, and to more than double the number of comments, by developers. This significant increase in the bug tracking activity is all the more remarkable given that two (out of nine) developers highlighted that it was hard to adapt to the new bug tracker, and thus they reduced their activity after the switch.

We not only observed an increase in activity by developers but also an increase in the number of distinct developer reporters and commentators. This shows that developers are active more often.

When we asked developers how they would interpret this effect on the level of activity of developers, they almost all (7 out of 9) mentioned the more comfortable experience of using GitHub's integrated bug tracking system, reminding many of the motivations they had previously listed for switching (better interface, less slow, same location for issues and pull requests, etc.). Many developers told us that they personally did not always open issues on the Bugzilla bug tracker when they found small bugs, and they now did open issues more often because it was easier to do so. One developer in particular admitted he had not been very active on the Bugzilla bug tracker since the main development activity had moved to GitHub's pull requests.

In addition, the GitHub notification system (including the mention system which allows attracting a developer's attention on a specific issue) was identified as being responsible for part of the additional activity. When a user is subscribed to a repository (which is the case of most Coq developers), notifications (by e-mail or in the web interface) are sent whenever a new issue is opened or a comment is posted. This can attract the attention of developers more quickly and lead to more comments in response. It was mentioned how some developers had trouble adjusting their settings to make it easier to manage this large flow of information (GitHub's notification default settings are quite unsuited to repositories with large levels of activity).

4.4.5.2 RQ2: Impact on quality of bug tracking activity

Measuring quality of bug tracking activity is difficult. That is why we could only answer this second research question qualitatively, by asking developers if they perceived a change in quality of issue reports and interactions on the bug tracker. A majority (5 out of 9) claimed that GitHub helped at least some reporters produce better issue reports (thanks in particular to formatting and the issue template[4]). However, three developers added that the switch had also attracted less experienced users whose reports were sometimes of a lower quality.

Regarding the quality of interactions on the bug tracker, again a majority of developers (6 out of 9) claimed that it improved after the switch. In particular, it was mentioned that users are more reactive and answer more frequently when they are asked additional information.

Several developers also emphasized that it was easier to discuss on the new platform, especially with respect to bugs that cross project boundaries. Indeed, GitHub makes it easy to link between issues, not only in the same repository, but also across the repositories of multiple projects. Examples of related projects include projects that Coq depends on (the OCaml compiler [176], the Dune build system [81], the Sphinx documentation tool [39], etc.) and projects that depend on Coq (in particular the Coq plugins and libraries that are tested in CI, cf. Section 3.6.2, but also editor support packages like the Proof-General [16] and Company-Coq [229] major and minor Emacs modes, etc.). Ma *et al.* [180] have specifically studied the process of fixing bugs that cross project boundaries, in the context of the scientific Python ecosystem.

Other perceived quality improvements include an easier navigation among issues, which helps both developers do a more efficient triaging, and users discover existing issues more easily, on which they can subsequently leave comments adding information. This participation of users is useful to improve the overall quality of some preexisting reports.

[4]GitHub allows setting up a template to help users opening issues know what information they should provide [31] (similar to the pull request template that I presented in Section 3.5.2). The issue template was added the day of the switch: `https://github.com/coq/coq/pull/1149`.

4.4.5.3 RQ3: Impact on the audience of the bug tracker

In Section 4.4.3, the clearest change after the bug tracker switch was the increase in the number of distinct weekly non-developer commentators. This observation was further confirmed in Section 4.4.4: the estimated impact of the switch was to more than double the number of distinct non-developer commentators in an average week.

When asked to interpret this surge, all the developers said that GitHub was more accessible, either because it is a very popular platform on which many people already have an account (especially computer science students who represent a large proportion of Coq users), or because it is quite simple to use, especially in comparison with Bugzilla. These explanations apply to a general increased participation of non-developers, but do not explain specifically an increase in the number of distinct commentators (as opposed to reporters).

Some factors that were mentioned by a few developers and could have specifically influenced the number of commentators are that some external contributors who already contributed through pull requests now also comment issues more systematically, that more attention is given to newcomers, that there is generally more discussion between developers and non-developers on the bug tracker, and that it is easier for interested users to subscribe to just part of the bug tracking activity (and thus participate in the discussion without following the activity on the whole bug tracker).

4.4.5.4 General discussion

Beyond the scope of our research questions, we gathered other perceived effects of the bug tracker switch that affect the quality of software development in a broader way. Most of the developers we interviewed noted that the bug tracker switch to GitHub contributed to a more general culture switch towards a more recorded communication and transparent development. In particular, it was mentioned that GitHub makes written communication less costly, that the mention system can be used in place of personal e-mails, which avoids losing technical content in private threads, and that more types of discussion take place in the bug tracker (included related to long-term plans or development processes).

In addition, some developers found that GitHub provides a more global view of the state and history of development. They also perceived an increased visibility of developers' own work, and of the Coq project overall.

While our RDD methodology can only be used to measure the immediate causal impact, an important question concerns the long-term effect of the switch. Our descriptive statistics (Section 4.4.3) and robustness checks on longer time horizons (Section 4.4.6.1) seem to validate that the observed effect of the switch was durable. This can be explained in part because GitHub evolves faster, according to the needs of ever-evolving open source developing processes. During our interviews, we learned that some of the recently added features of GitHub, like the automatic search for related issues [115], are particularly appreciated.

4.4.6 Robustness checks and threats to validity

4.4.6.1 Internal validity and robustness checks

The most important threat when performing RDDs, especially on time-series, is to have another event that occurred near the cutoff, and which could have influenced the outcomes. In our case, there were no such events that we could identify, except the 8.7.0 release, that took place on October 17th, just one day before the switch. To confirm that it is not the release that is driving our results, we perform some placebo robustness checks, i.e. we do the exact same analysis that we do for the bug tracker switch, replacing the cutoff date by the date of six other releases (all the other major releases since 2009, except the latest one because we do not have yet enough data points after the release). We show that the only positive and significant effect in the bug tracker activity after any proposed cutoff is the effect after the switch. For some outcomes, we observe a significant effect after some major releases but this effect is negative and of a much lower absolute magnitude. It is thus very unlikely that our results are driven by the release close to the switch and not by the switch itself.

Our main specification for RDD uses a small bandwidth and a linear model. As a robustness check, we estimate our model using larger time horizons (511 days, i.e. 73 weeks, on each side of the cutoff, corresponding to all the data that we collected after the switch and the same number of days before). To reduce the associated bias, we include a quadratic term to allow the regression to better fit the data and "adjust" to far away data points. Virtually all our results still hold with this specification, with comparable estimated magnitude, and with statistical significance (with only the number of distinct developer-reporters going just above the arbitrary 0.05 p-value threshold).

Temporal RDDs have some notable differences from other types of RDDs, as noted in a recent paper by Hausman and Rapson [130]. One major difference is the capacity of anticipation and adaptation to the treatment, that affects directly the outcome. To ensure that our results are robust to these effects, we additionally estimate donut RDDs (i.e. we exclude the data points just after and just before the switch, which are potentially problematic), as recommended in [130]. Our results still hold with comparable estimated magnitudes, and a few of them going just above the arbitrary 0.05 p-value threshold (this is not unexpected, given that we have removed data points, and this automatically increases the standard errors). Time-series are also subject to potential serial auto-correlation of errors (errors at time $t+1$ are not independent from errors at time t). Failing to take such auto-correlation into account can lead to underestimate standard errors. To check that there is no serial auto-correlation in our data, we plot residuals[5] [294], and notice the absence of any specific pattern beyond those which show the presence of heteroscedasticity, which we already control for (see the corresponding Jupyter notebook [323]).

[5]Residuals are the differences between measured and estimated values.

4.4.6.2 External validity

This is only a case study on a very specific project with a relatively small core team, and therefore it is very important that this study be replicated before we can draw general conclusions that can serve to formulate recommendations for open source developers. Our reusable assets, such as the migration tool and the analysis pipeline should be helpful to perform such replications.

4.5 Conclusion and perspectives

I have conducted the switch of the Coq bug tracker from Bugzilla to GitHub, including the migration of preexisting issues, and I have improved existing tooling in the process. Other open source projects have already benefited from this enhanced tooling (Ledger, migrating from Bugzilla to GitHub in 2018), or the experience that I drew from it and shared in a blog post (OCaml, migrating from Mantis to GitHub in 2019).

By analyzing mined repository data, we have shown that this bug tracker switch resulted in an increase in bug tracking activity in two ways. First, main developers are using the bug tracker more, by opening more issues and communicating more. Second, external contributors are more numerous to take an active role by commenting issues as well.

We complemented this quantitative analysis by interviewing the main developers. Almost all of them were very happy with the bug tracker switch. They provided us with insights to explain our estimated causal effects in bug tracking activity, and on the more general impact on the development process and its transparency.

Finally, we presented in details an example of the use of RDD, an econometric method to derive causality, that is seldom used in empirical software engineering.

Given the lack of literature on the effects of bug tracking environments, interested researchers could expand our analysis in many ways. Analyzing other outcomes (such as pull requests or the rapidity of bug resolution) could give interesting additional insights, and help create a more complete picture of the mechanisms at play. Replicating our analysis on more projects would increase the external validity of the results, and allow researchers to offer clear takeaways to projects interested in switching their bug trackers. Based on this first study, it seems that moving to a more popular, and integrated platform can help increase the level, the quality and the audience of bug tracking activity.

CHAPTER 5

FORMALIZING AND AUTOMATING THE RELEASE PROCESS

5.1 Introduction

The process of releasing new software versions is a crucial part of software development that directly affects how users will be able to benefit from it. It is related to the questions of compatibility between versions, of reactivity of bug fixing (because a bug fix is only useful to most users when it is eventually released), and of documentation (so that users can read about the things they need to be aware of before upgrading). Software that is not released for a long time will become incrementally less useful to its users [175] and may be viewed by them as dead. Software that receives large breaking changes, or where some of the breaking changes are not documented, can prevent users from upgrading, or at least significantly delay the process. Finally, well-thought and understandable version numbering schemes are as important as documentation [95, Chapter 7] in a world where software projects have more and more dependencies [77].

As observed by Michlmayr [195], release management is generally not an issue for young projects. They tend to release frequently, because getting user feedback is critical, and the release process is generally simple. Today, lots of conventional wisdom and tooling is available to help them: standardized versioning semantics [232], a standardized changelog format [171], automatic changelog generators [58], continuous integration and deployment services [15, 51, 99, 118, 196], release notes and artifact sharing mechanisms [214], ecosystem-specific package managers (cf. Chapter 6), etc.

However, as projects mature and grow larger, releases tend to be produced less frequently. The seven projects that Michlmayr studied in his thesis [195] had all experienced significant delays in producing releases before they decided to switch to time-based release cycles.

Longer time between releases creates a number of problems, and a switch from feature-based to time-based release cycles is generally expected to solve them, resulting in: shorter time for users to benefit from new features and fixes, and thus shorter feedback loop for developers, less disruption between two successive releases, better planning and no “last-minute feature rush” [95] (or “thundering herd of patches” to reuse the expression of kernel developer Ted Ts’o [60]) because missing a release is not a big deal if the next one is coming shortly after.

However, switching to short release cycles comes with a number of associated challenges, to make the release process efficient and lightweight enough. After a historical presentation of the switch to time-based releases in the Coq project, I present some challenges that this created, and how I contributed to solve them. Beyond the challenge of continuous testing that was addressed in Section 3.6, the three main challenges presented in this chapter are: how to separate the contributing process from release management; how to streamline the backporting process; how to prepare consistent release notes.

While the complex release strategies of large open source projects are already well documented [89, 108, 152, 164, 236], and short release cycles have now been largely studied [49], my contribution should be useful to medium-sized projects, which have overgrown what standard recommendations and tools can help them do, without having reached the size nor requiring the process complexity of the biggest open source projects which have been the focus of most research so far.

5.2 Historical overview: towards more frequent releases

Until the release of Coq 8.5, the frequency of new releases used to vary a lot (cf. Table 5.1), depending on the features that were planned for the release: Coq 8.5 was released three and a half years after Coq 8.4, which was released two years after Coq 8.3, which was released one and a half year after Coq 8.2. The period of beta testing, separating the first beta release from the final release was frequently above six months, even rising to a full year in the case of Coq 8.5.

Under the impulse of Maxime Dénès, Coq adopted time-based, shorter release cycles, starting with Coq 8.6, which was released 11 months after Coq 8.5, Coq 8.7, which was released 10 months after Coq 8.6, and Coq 8.8, which was released just 6 months after Coq 8.7. An official release manager was appointed (initially Maxime Dénès; and since Coq 8.9 it has become a rolling position); feature freeze dates were set by the release manager, instead of waiting for the completion of the planned features; beta releases were shortened thanks to continuous testing of the master branch (cf. Section 3.6).

Coq has also adopted dot-separated three-digit version numbers corresponding to the standard scheme [95, Chapter 7], which is used in particular in semantic versioning [232], except that it does not (at least it does not yet) follow the semantics of semantic versioning (both first

Major releases	Release dates	Time since previous release	Duration of beta-testing
8.2	2009-02	–	8 months
8.3	2010-10	20 months	8 months
8.4	2012-08	22 months	8 months
8.5	2016-01	41 months	12 months
8.6	2016-12	11 months	1 month
8.7	2017-10	10 months	1 month
8.8	2018-04	6 months	1 month
8.9	2019-01	9 months	2 months
8.10	2019-10	9 months	5 months

Table 5.1: Dates of major releases since 8.2, with final release dates, elapsed time since the previous final release date, duration between the first beta release and the final release (the stabilization period is in fact longer). Source: `https://coq.inria.fr/news/`.

numbers indicate major releases, there are no minor releases, the last digit does indicate patch-level releases).[1] Patch-level releases have also been released more frequently (every two to four months) since Coq 8.7.

Overall, the adoption of shorter release cycle has successfully managed to provide more stable versions, with less breaking changes, that are easier to upgrade to (see also the compatibility policy described in Section 3.6.3.1). In-depth evaluations of the time it takes for feature or bug fixes to reach users, similar to what was done by da Costa *et al.* [68], or of the cumulated effort users invest in porting their projects to new versions of Coq over many years, would be worthwhile, but have not been conducted as of today. However, the switch to shorter release cycles also imposes efficiency constraints, and this is what this chapter is about.

5.3 How to separate contributing from release management?

5.3.1 Branching strategy

Branches are a wonderful feature of modern version control systems (VCS) that allow project owners to manage several versions of a code base in parallel. We generally distinguish two kinds of branches. Topic branches are short-lived branches that developers use to work on a specific topic (such as a new feature or a bug fix): they are the base ingredient of pull-based development. On the other hand, a limited number of long-lived main branches (at least one) is maintained by a project to manage different versions of the software: one of them is usually distinguished and called “master” in the Git terminology, or “trunk” in the SVN terminology. In this section, I am focusing specifically on how these main branches are managed.

[1]The scheme that was used previously in the Coq 8 series was 8.XplY where 8.X was the major version number, and Z was the patch-level number. This scheme had been adopted from other projects (in particular PHP projects) using it (source: `https://github.com/coq/opam-coq-archive/issues/8`).

In small open source projects, it is quite frequent to have a single maintainer that takes care of the integration of pull requests and the preparation of new releases, or to have a small team of maintainers that work together to make sure that changes are integrated, and new versions are released according to a shared plan. For small software projects, such as software libraries, releasing a new version may be as simple as publishing a new tag and a changelog. In these cases, branch management may be easy: the project maintains a single public branch, usually called master, from which all new versions are tagged. Even patch-level releases can be prepared from the master branch, as long as no new feature has been integrated since the last release.

Because releases are cheap, patch-level releases frequently include just a few bug fixes (sometimes just one). Besides, because minor releases are easy to upgrade to, when the project follows semantic versioning, maintainers will usually not bother releasing a patch-level version for a fix that has been integrated *after* a feature, unless the bug is considered of critical importance (typically security fixes).

In a study of 2,923 GitHub projects (selected for having five-year of active history, at least 100 commits, 3 contributors, one closed pull request, and excluding forks and non-software projects), Zou *et al.* [328] found that 18% of selected projects only use a single branch. Among the projects that use multiple branches, they found that about 25% of the non-master branches are used for version iteration (i.e. what is usually designated as *release branches*), the rest can be considered topic branches. Unfortunately, they did not compute the proportion of projects that use release branches.

For larger projects with many contributors, new features are prepared, reviewed, and ready to be integrated all the time. Stabilization in preparation of a new release may require coordination through a feature freeze (a period during which no new features may be integrated). This is what is done in the GCC project for instance [108]. A first release candidate is published at the end of the feature freeze. At this time, a new release branch is created. This release branch will make it possible to prepare patch-level releases without preventing the integration of new features in the master branch.

In the Coq project, in order to make it possible to involve a larger group in the pull request integration process (cf. Section 3.7), we proceed slightly differently to handle the stabilization phase leading to a first beta release. What we call erroneously a feature freeze is in fact a cutoff date at which time a new release branch is created (from the current state of the master branch). This release branch will be where the stabilization phase will take place, under the authority of the release manager, which is the only person allowed to update this branch (and whom we should call the “release owner” according to Fogel’s definition [95, Chapter 7]). This means that the integration of new features in the master branch never stops, so integrators do not have to care or even be aware of the release schedule. Meanwhile, it may take a month or more before a first beta version is finally released. No release is ever done from master.[2]

[2]Although the master branch still receives specific “alpha” tags marking the first commit to *not belong* to a given

This workflow was made possible by a change in branch management, that occurred previously under my conduct.

5.3.2 Inverting the patch-porting workflow

The previous model (that was used from January 2015 to July 2017) was similar to the one that is used today in the Symfony project [270, Section "Choose the right Branch"], except that it was not formalized nor documented. Bug fixes had to target a release branch (the earliest branch in which the developer wanted the fix to be applied), and release branches were subsequently merged into more recent release branches, all the way up to the master branch. This process had several issues: bug fixes took longer to reach the master branch, which some users used as their working version of Coq; and contributing was made more difficult by having to choose on which branch to prepare a pull request.[3] Despite the thorough contributing documentation that Symfony provides, they suffer from the same issue.[4]

The new process uses backporting [140] instead: pull requests should virtually always target the master branch, and they are cherry-picked (i.e. automatically reapplied) to stable release branches when they satisfy the criteria for inclusion. This process solves the issues of the previous model: fixes are immediately available in master, and contributors do not have to worry about which branch to target. Furthermore, because the backporting process occurs *after* a pull request has been merged, the release manager can reject some proposed changes without requesting anything from the pull request author. Thus, it makes it less likely that they will have to bear a debate on the acceptability of the patch every time this happens, nor have to sound dictatorial.

The new process was introduced at the time the release branch for Coq 8.7 was created. From this time, all pull requests should be merged in the development branch, unless they are specific to a release branch. Only the person in charge of the release branch (initially myself for the release branches of Coq 8.7 and 8.8, a period during which Maxime Dénès and I shared some responsibilities of release management, then the appointed release manager starting with Coq 8.9 when it became a rolling position) can push to the release branch.

stable version, and thus the beginning of the development of the next one.

[3]See for instance `https://github.com/coq/coq/pull/463`, `https://github.com/coq/coq/pull/495`, `https://github.com/coq/coq/pull/538` (bug fixes that initially targeted the master branch) and `https://github.com/coq/coq/pull/90`, `https://github.com/coq/coq/pull/191` (features that initially targeted a release branch).

[4]Here is a hand-picked selection of pull requests that did not target the right branch from the start, and had to be rebased from August 5th to August 16th, 2019: `https://github.com/symfony/symfony/pull/32947`, `https://github.com/symfony/symfony/pull/32977`, `https://github.com/symfony/symfony/pull/33093`, `https://github.com/symfony/symfony/pull/33151`, `https://github.com/symfony/symfony/pull/33199`.

5.4 How to streamline the backporting process?

5.4.1 Visualizing the backporting tasks

A common issue in projects that rely on backporting, especially in the absence of a clear and formalized process on what gets backported and who does it, is that some commits get backported when they should not have, and some commits do not get backported when they should have. A previous study on backporting in the Linux kernel [140] found that, while only experts are tasked with backporting activities, a significant proportion of them still finds it a hard and error-prone process. Recently, machine-learning tools have been designed to select patches for automatic backporting [86, 134, 301], but it has sometimes led to patches introducing regressions getting backported nonetheless [61].

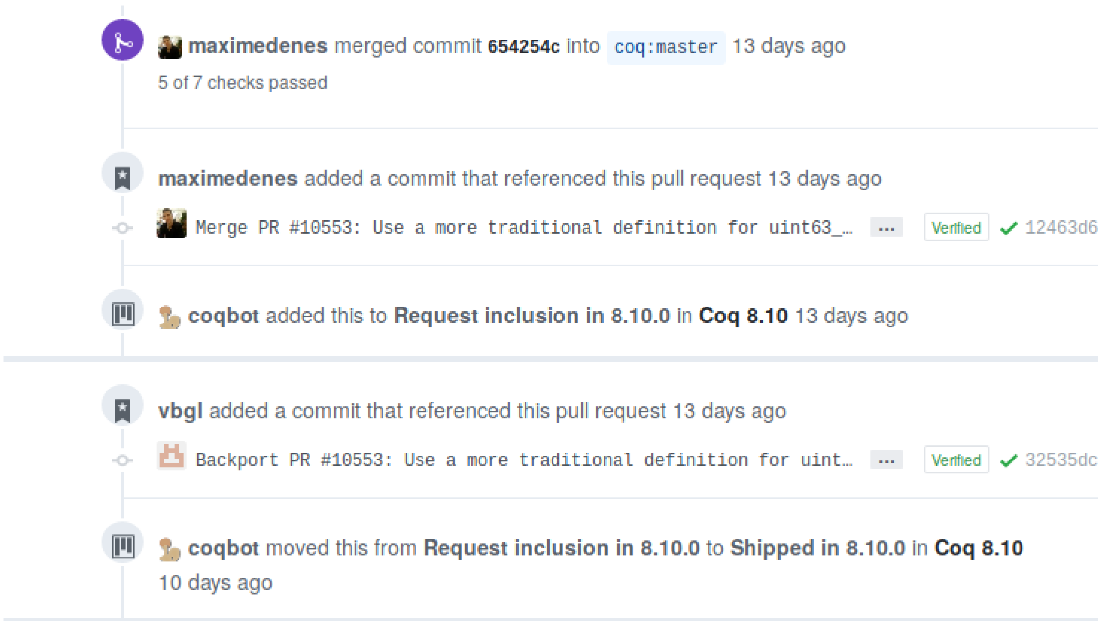
I implemented a process whose goal is to avoid these kinds of issues, and which I adapted based on the feedback I received during the first few weeks of its application.[5] The process relies on milestones, and a “GitHub project”. “Projects” are a GitHub feature that was announced in 2016 [295], which allows to create a Kanban-like board with columns that can contain notes, issues, and pull requests.

Before merging a pull request, the pull request assignee must choose a milestone (and is reminded to do so by the merging script, see Section 3.7). Most of the time, the choice is between the upcoming version of the release branch (which may be a beta version, a “final” version, or a patch-level version) and the version under active development. This gives an indication of intent, but ultimately it is the decision of the release manager to follow the indication, or to edit the milestone. It is similar to the tag `stable@vger.kernel.org` that patch authors or integrators can use in commit messages to signal backporting intent to stable kernel maintainers.

When a pull request is merged into the milestone corresponding to the next stable release, the pull request is added to a “Request inclusion in ...” column in the project board. Then the release manager selects some pull requests to backport, and does it using a backporting script (that automatically cherry-picks the pull request commits onto the release branch). The batch of backported pull requests is generally pushed to a branch so that CI can run. When backported pull requests are finally pushed to the release branch, they are moved to a “Shipped in ...” column.

The use of a GitHub project board makes the backporting process both non-invasive, and transparent. Kerzazi and Robillard [163] also report on the use of a Kanban-inspired release process, and although the use of the Kanban board is different (it is used actively by multiple developers, and is used for the whole release process instead of specifically the backporting process like here), one of their first motivating issue, “lack of visibility on the release process”, is similar.

[5]Cf. `https://gitter.im/coq/coq?at=596dfff82723db8d5e217283` and `https://gitter.im/coq/coq?at=598c0281329651f46e079a36`.


maximedenes merged commit 654254c into coq:master 13 days ago
5 of 7 checks passed

maximedenes added a commit that referenced this pull request 13 days ago
Merge PR #10553: Use a more traditional definition for uint63_… … Verified ✓ 12463d6

coqbot added this to Request inclusion in 8.10.0 in Coq 8.10 13 days ago

vbgl added a commit that referenced this pull request 13 days ago
Backport PR #10553: Use a more traditional definition for uint… … Verified ✓ 32535dc

coqbot moved this from Request inclusion in 8.10.0 to Shipped in 8.10.0 in Coq 8.10
10 days ago

Figure 5.1: A screenshot of a pull request getting merged, automatically added to the backporting project, and getting backported by the release manager (of Coq 8.10).

5.4.2 Automation

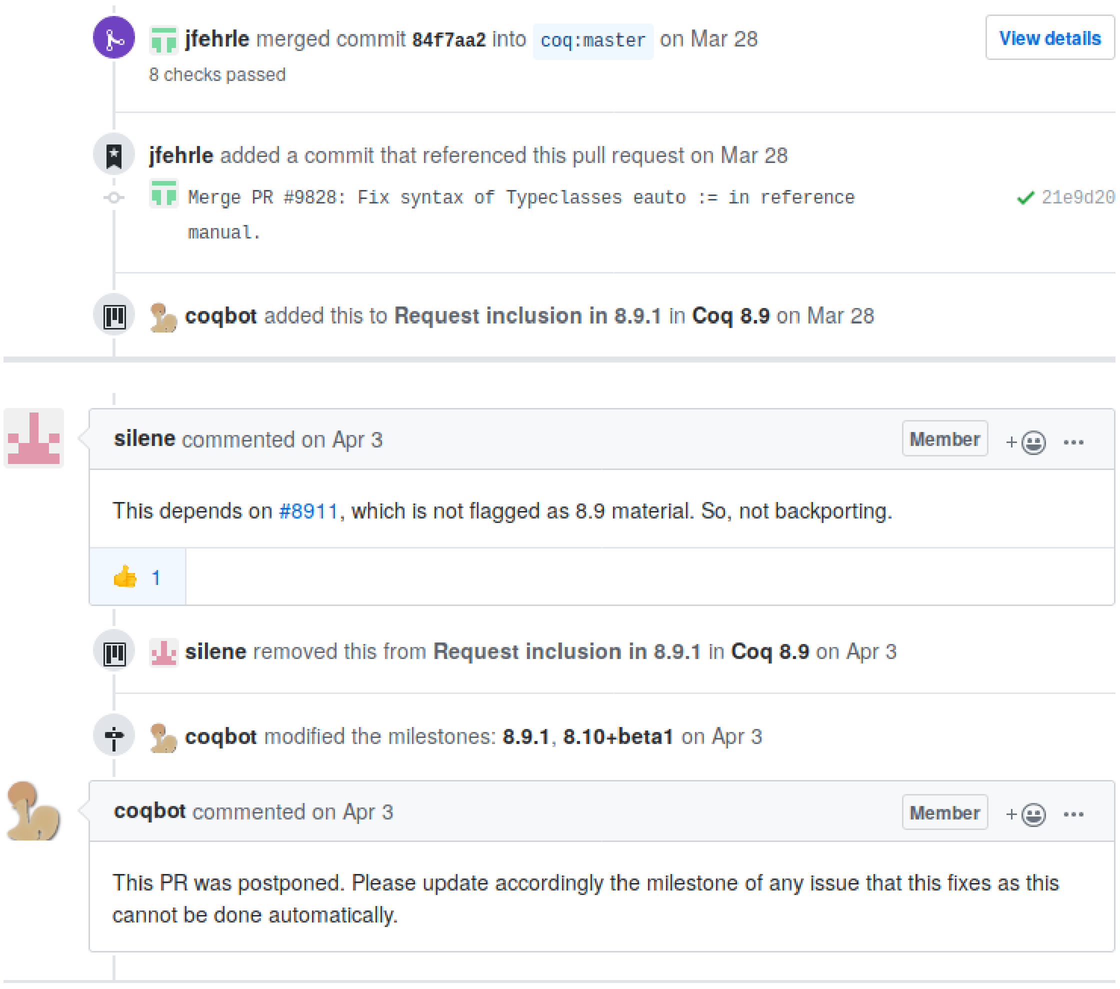
At first, I used this process and managed the project board manually, using filters to find which pull requests needed to be added to the first column. Then GitHub added simple automation to project boards [204], which despite being too simple for the kind of workflow I was using, still inspired me to automate the management of pull requests within the backporting board. I implemented this automation in the bot (presented in Section 2.5).

It is triggered by any pull request, push, and project board event. When a pull request is merged on master, it checks whether the description of its milestone contains an indication of a column to add the pull request to. If it is the case, it adds it there. When a pull request is backported to a release branch, it looks again at the description of the milestone to see to which column it should move the corresponding card (see Figure 5.1).

Finally, when a pull request is removed from a project board, it checks the milestone description to see if it was removed from a “Request inclusion ...” column corresponding to its milestone, and if it is the case, it updates the pull request milestone to the one of the version under development, and posts a message informing the stakeholders that the pull request was postponed (see Figure 5.2).

5.4.3 Open issues, future work

The current workflow has a known limitation that will need to be addressed in the future. Some more automation could also be implemented.


jfehrle merged commit 84f7aa2 into coq:master on Mar 28

8 checks passed

View details

jfehrle added a commit that referenced this pull request on Mar 28

```
Merge PR #9828: Fix syntax of Typeclasses eauto := in reference
manual.
```

✓ 21e9d20

coqbot added this to Request inclusion in 8.9.1 in Coq 8.9 on Mar 28

silene commented on Apr 3 — Member

This depends on #8911, which is not flagged as 8.9 material. So, not backporting.

👍 1

silene removed this from Request inclusion in 8.9.1 in Coq 8.9 on Apr 3

coqbot modified the milestones: 8.9.1, 8.10+beta1 on Apr 3

coqbot commented on Apr 3 — Member

This PR was postponed. Please update accordingly the milestone of any issue that this fixes as this cannot be done automatically.

Figure 5.2: A screenshot of a pull request whose backporting is rejected by the release manager (of Coq 8.9). In this case, the release manager took the time to explain the decision, but when that is not the case, the automatic message from the bot is useful so that stakeholders are informed.

The main limitation to the use of milestones to suggest backporting is that a pull request can have at most a single milestone. The presented workflow is thus not well suited to maintaining several release branches at the same time. Currently, the Coq project does not maintain old stable versions (producing new patch-level releases after a more recent stable version has been released). However, it can happen that the last patch-level release of the current stable version is still in preparation while the branching and stabilization process for the next stable version has already started. In this case, it can happen that a pull request should be backported to the two active release branches.

Up to now, such cases have been infrequent enough to be managed manually. However, formalizing and automating this part of the workflow as well would ensure that accidental mistakes do not occur. In particular, a mistake would be for the release manager of the current stable version to backport a fix and publish a new patch-level release which incorporates it, but for the release manager of the upcoming stable version to miss this fix and not backport it. This would then create a regression, because users updating from one version to the next (both semantically and chronologically) would discover that the bug that was fixed is back.

Possible improvements to the automation include: creating automatically the backporting project board; managing the case of pull requests that were not marked for backporting but the release manager decides to backport nonetheless; not posting an automatic comment when the release manager has rejected a pull request but has just posted a comment explaining why; automatically running the backporting script for the pull requests that can be cherry-picked without conflicts, so that the CI results are already available, and the release manager can just approve or reject them.

5.5 How to prepare consistent release notes?

5.5.1 Changelog and release notes in software development

Keeping a changelog is considered to be an important quality criterion in software development. Comprehensive and understandable release notes are especially useful to preexisting users of a software product, as it will help them understand what differences they should expect in the new version, especially what are the incompatibilities, the new features, and the bug fixes (possibly only the most important ones if the list is too long).

Several attempts have been made to formalize how a changelog should be kept [98, 171]. In particular, the "Keep a Changelog" initiative, started in 2014, and whose version 1.0.0 was published in 2017, has been quite successful among projects hosted on GitHub, since searching for files whose name is exactly `CHANGELOG.md`, and which contain the words "The format is based on Keep a Changelog" yields 200,000 results.

There are also initiatives to automate the generation of release notes based on commit messages that should respect a conventional format [58]. However, a major issue with automatic changelog generation is that it usually happens at release time only, which makes it harder to support users that use the master branch as their working version.

Finally, we should note GitHub has native support for attaching release notes (and release artifacts) to a git tag since 2013 [214].

5.5.2 Three issues with Coq's changelog

Mutliplication of information sources The Coq project has maintained a changelog since at least Coq 6.1, which was released in 1996.[6] However, there also existed in parallel:

- a `COMPATIBILITY` file, which was created in 2006,

[6] See `https://coq.inria.fr/distrib/V6.1/V6.1-ocaml1.07.tar.gz`.

- a "Credits" chapter in the reference manual (since 1995), which contained less detailed release notes that, in addtion, credited the authors of the main changes, and listed the contributors of each version,
- a news section on the website where new releases were announced even more succinctly,
- the GitHub repository's release page, which is used since Coq 8.7 to publish binary installers, and also contains a brief description of what is in the release.

The duplication between these various sources was an issue that created more work for the release manager, and risked creating inconsistencies.

The two other issues which the project's changelog suffered from are not specific to the Coq project, and are likely to arise in any project receiving numerous contributions.

Misplaced changelog entries When a pull request adds a new entry into the changelog, but the integration process takes longer than expected, and the pull request ends up missing the upcoming release, it can happen that what was the "Unreleased" section of the changelog file has become a released section (the title of the section has been changed, and a new "Unreleased" section has been created), and yet the pull request has no conflicts (or at least no conflicts in the changelog file). If the author and the reviewers of the pull request do not realize this, and forget to update the location of the changelog entry, the pull request can eventually get merged with a new entry in an already released section, and this can be hard to notice later on.[7]

But such issues happened even more frequently after we put the backporting process in place, since it can happen that the decision of the release manager to backport or not does not correspond to the expectations of the pull request author and reviewers.[8]

Frequent conflicts in the changelog file Because a lot of pull requests developed in parallel must add an entry to the same changelog file, it is frequent to encounter merge conflicts in this very file. However, this issue did not arise frequently enough that it was considered a problem that needed to be fixed (the changelog of Coq is divided in categories for various topics, and this simple division reduced the number of such conflicts).

Some other projects, for instance GitLab, have suffered from this issue much more [261].

5.5.3 Proposed solution, alternatives

The first step of the solution is to separate the release notes and the unreleased changelog. This step alone can solve the issue of misplaced change entries. It was recently adopted by the Math-

[7]For instance, here are two commits fixing the location of a change entry: `https://github.com/coq/coq/commit/594e53cda3845794bf1e14aec7b0d2a5ee9cd075` (noticed and fixed before merging), `https://github.com/coq/coq/commit/9468bcd39808f4587d3732f46773b1e339b2267c` (noticed only after merging).

[8]Here are three pull requests fixing such issues: `https://github.com/coq/coq/pull/1010`, `https://github.com/coq/coq/pull/8354`, `https://github.com/coq/coq/pull/8926`.

Comp project [56]. Although they are otherwise following the convention from "Keep a Changelog" [171], they have split the "Unreleased" section to a new file: `CHANGELOG_UNRELEASED.md`.

The second step of the solution is to split changelog entries to multiple files. The purpose of this step is to avoid merge conflicts that occur when everyone is modifying the same file. This is the same solution we had used to avoid the (much more frequent) conflicts that occurred with overlays (cf. Footnote 27 in Section 3.6.2).

Then the release manager needs to be able to compile all the relevant changelog entries when preparing a release. But the solution is simple, and was highlighted in a post on GitLab's blog explaining how they solved their "CHANGELOG conflict crisis" [261]. When a branch is used to prepare a release, all the changelog entries that are present in this branch are the ones that need to be selected. A script can be used to compile all the selected entries together, add them to the release notes with the right version number, and remove the corresponding files from all the branches that contained them (either by applying the change first to master and then backporting it to the release branch, as we do in the Coq project, or by applying the change to the release branch and merging to master, for projects that prefer this way of working).

The last steps for the Coq project were to consolidate the changelog, the Credits chapter, and the `COMPATIBILITY` file to a single place, which is a chapter of the reference manual. To properly support users that use the development branch as their working version, the unreleased changelog entries are compiled together in an "Unreleased" section of this chapter, and we continuously deploy the reference manual to the web. A boolean variable determines whether this "Unreleased" section appears or not, and in case it does not, a test file ensures that there are actually no unreleased changelog entries in the branch. This test is useful to avoid a situation where a pull request would be backported at the last minute *after* the release notes have been generated and inserted, and whose changelog entry could be forgotten.

An alternative to this solution is to automatically generate the changelog. It is actually a pretty similar solution, except that the source of information are not individual files describing the changes. Instead, the information can be extracted from commits [58], or merged pull requests and closed issues [169]. The main advantage of using a committed file instead of commit messages, pull request titles, or descriptions, is that the changelog entry is now part of the code, and gets reviewed with the rest of the pull request, leading to more polished release notes. In the case of the Coq project, by making the changelog a chapter in the reference manual, we can even use rich-text markup to link to relevant sections of the documentation.

5.5.4 Open issues, future work

While these changes have already solved the three issues listed in Section 5.5.2, this is still a work-in-progress that I intend to complete.

First, while the consolidation of the changelog entries into a single "Unreleased" section in the reference manual is already automated in the build system, the preparation of a new release still involves manual steps: the manual on the release branch has to be built, so that the "Unreleased" section can be extracted and inserted in the relevant location, with an appropriate title, the changelog files present in the release branch have to be listed and deleted, a pull request targeting the development branch has to be opened, merged, and backported. A dedicated script, or the bot, could be used to automate part of these steps.

Second, the documentation for adding a new changelog entry shows a specific structure, with a change description, a link to the corresponding pull request, possibly the closed issue, a list of contributors (author, co-authors, and reviewers when their contribution is significant). Experience gathered in the few months following the introduction of this new workflow has shown that contributors were not following the proposed structure rigorously, and this is not something that reviewers can enforce either (especially as it would amount to nitpicking). To simplify the process of adding a new changelog entry, Gaëtan Gilbert introduced a simple shell script that asks basic questions and uses this information to generate a new file, to be edited by the pull request author. This script alone has helped contributors provide more consistently structured changelog entries. An alternative would have been to move from an unstructured changelog file (currently a simple `.rst` Sphinx [39] file) to a structured file (for instance using YAML syntax [25]) that would contain a short description, an optional long description, the list of authors, reviewers, pull request and issue numbers, and from which the actual changelog entry would be generated. We will see in the future if the need for such structured files arises, or if the script to generate changelog entries is sufficient.

Finally, yet another file in the Coq repository contains duplicated information: the `CREDITS` file. Its purpose was to gather the complete list of contributors, and their years of contribution, but in practice, it has been scarcely maintained, especially since the switch to a pull-based model. In the previous version of the contributing guide, I had included a note that contributors should insert their name and the current year to this file, and update the years of contribution when appropriate, but this has not been sufficient (less than a dozen contributors have followed this advice). This led me to remove this suggestion when I refactored the contributing guide (cf. Section 3.4). In the future, we should either find a way to make sure that this file is kept up-to-date (for instance by using a checklist item in the pull request template, cf. Section 3.5.2), or remove this file entirely (given that it is partly redundant with the changelog entries, that already contain contributor information).

5.6 Conclusion and open issues

In order to facilitate the release process and make it easy enough to make the release manager role a rolling position, a number of steps have been implemented. I have already listed a number of them in the previous sections. We changed branch management, from a model where contributors had to decide which branch their pull requests target, and where bug fixes were merged from release branches into the development branch, to one where all pull requests should target the development branch, and it is the role of the release manager to backport relevant changes to the release branch. Both models are commonly used in open source projects, but the new one presents a significant advantage which is to make the contributing process more straightforward. This however puts more responsibility on the shoulders of the release manager, so I have introduced a process based on a GitHub project board to efficiently manage the backporting tasks, and I have automated part of it. Finally, I have identified a number of issues related to the preparation of release notes, so I have introduced a new workflow and associated tooling to solve them.

Other steps have been taken but were not mentioned so far. Package generation and signing has been mostly automated. The documentation is automatically deployed, and the release artifacts are published directly on GitHub, which means that release managers do not have to worry about having access to the website server. I have documented the release process in the contributing guide, and I have created a checklist that release managers can use to ensure they follow all the steps of the release process [326]. As already mentioned in Section 3.5.2, checklists have been advocated as underused and underappreciated memory aid to increase success in many areas [105, 216, 219].

This checklist will serve as a basis to identify even more parts of the release process that can be automated (some elements of improvement have already been mentioned in Sections 5.4.3 and 5.5.4).

An open question is what should the ideal release cycle duration be. The target duration was nine months when time-based releases were introduced, then six months starting with the 8.8 release (although this target has actually been matched only once so far). However, six months is still a long release cycle compared to other projects (like Google Chrome and Mozilla Firefox [68, 236]), and a long time for a contributor to wait until the feature they contributed becomes available to all (for bug fixes and documentation, the wait is not so long because these usually get backported to patch-level releases). Therefore, the Coq project still suffers from feature rush when a feature freeze is approaching.[9]

Some developers would be supportive of a shorter release cycle, but besides an even more automated release process, this would require changing our compatibility policy (cf. Section 3.6.3.1), because we cannot reasonably expect users to port their projects (because of breaking changes) more than twice a year. A possible solution would be to adopt semantic versioning [232], and to start releasing minor versions, with new features but no breaking changes, more frequently (for instance every three or four months), while reducing the frequency of major releases including breaking changes (for instance to one every year).

[9]Although, there might not be any hope of avoiding feature rush completely: Rahman and Rigby [235] observed that even in projects with short release cycles such as Google Chrome and the Linux kernel, there is some degree of rush before release stabilization begins.

PART II

ORGANIZING COQ'S PACKAGE ECOSYSTEM

CHAPTER 6

Package distribution

6.1 Introduction

Efficient software engineering and the conception of large software systems critically rely on reusable code [253]. Since the rise of open source ecosystems and package managers, software reuse has become a systematic coding practice, even for small and medium-sized software projects. Decan *et al.* [77] found that a majority of packages depend on other packages in all seven ecosystems they studied.

Ecosystems of good, open source libraries can be the criterion that makes a programming language successful in general, or in a particular application domain (much more than specific language features [194]). For instance, this is what made me choose Python for my data analysis instead of OCaml, which is a language that I like more and have more experience with (cf. Section 2.4).

However, having a community producing good, reusable libraries is not enough: the distribution model of these libraries to the user community is of critical importance as it will directly impact how much effort is required for library authors to make their libraries available, and for users to reuse preexisting code, rather than building their own.

Libraries, or other kinds of reusable software artifacts such as plugins, extensions, and tools, are generally distributed as single units called "packages", through a "package manager". While package ecosystems have recently received much attention from researchers, academic research on the topic of package management is more limited, and is usually focused on technical solutions to very specific problems (cf. our list of academic papers studying package manager design, and package manager ecosystems [207]).

While a number of empirical comparisons of various package ecosystems have been produced in recent years [34, 35, 73, 74, 77, 80, 166], there is a lack of papers reviewing socio-technical options that package manager designers can choose from, or comparing socio-technical merits of existing package managers. And these options are numerous because the design space of package management has largely been explored by many software engineers, having built a great variety of solutions over the last three decades.

During this long period, some early problems, in particular the ones from a pre-web era, have become obsolete, and others have emerged as the practice of software reuse through package managers has grown to unforeseen proportions [215].

In this chapter, I attempt to sketch the landscape of package management, starting with some definitions, a few words of history, and a non-exhaustive list of socio-technical options. Presenting these options is important because, as noted by Mens [191]: “The main challenge for any software community resides in how to design and structure its ecosystem in such a way that it fosters a lively community, while at the same time maintaining the quality, stability and reliability of the ecosystem”.

Then, I present the history of package distribution in the Coq ecosystem, what unresolved issues remain, and some options to address them.

6.2 Definitions

Package A software *package* is a piece of software bundled in a way that makes it easy to distribute and reliably install. The most basic packages are tarballs containing source code and an *autoconf*-style [181], automatic install procedure. Packages can use a standard layout and provide standard meta-data expected by a specific package manager.

Manifest file When a package must provide standard meta-data, this is generally done through a manifest (or meta) file. This file typically contains information on the package dependencies, possibly with its author, homepage, bug tracker URL, license, version number, and build procedure.

Package manager A *package manager* is a software tool that can reliably install and remove a piece of software on a computer. One of the tasks of the package manager is to install all the required dependencies besides the requested software. Sometimes, such step will require making choices about the version of the dependencies to install, and this step alone can be highly complex [183].

Package registry A *package registry* is a central location for sharing software packages (i.e. not just software but also the required knowledge to properly install them), on which a package manager depends. The package registry may or may not host software source code, and precompiled binaries, besides basic meta-data such as dependencies, and, often, licensing information. Package managers frequently come with a specific package registry, but some package managers are mere user-side tools that depend on preexisting registries,[1] while some package registries are alternative locations from which to fetch (other) packages,[2] typically for use with a package manager that can support alternative registries.

Package repository or archive *Package repository* or *archive* are synonyms for package registry that are more suited when the entire content of the registry is stored as hierarchically-organized files, and can easily be mirrored, or forked. Sometimes, the package repository is a single git repository in which people can collaborate on package meta-data: examples include Homebrew (`https://github.com/Homebrew/homebrew-core`), MELPA (`https://github.com/melpa/melpa`), nixpkgs (`https://github.com/NixOS/nixpkgs/`), opam (`https://github.com/ocaml/opam-repository`)...

Package index A *package index* is a central location to search for existing packages. This is often strongly tied to (and thus confused with) the package registry. However, some exceptions make the distinction necessary: the package registry is usually machine-friendly, and a human friendly website may be needed on top, in which case, the indexes of two package registries based on the same technology might be quite different,[3] the package distribution method may be entirely distributed (with the package manager requiring URLs instead of package names), in which case there is no registry but an index can still be useful to discover what is "out there",[4] or the package registry might be so large that some restricted index is preferred.[5] Finally, even manually curated lists[6] [307] may be considered as package indexes.

6.3 A brief and incomplete history of package managers

Because package managers' origins are multiple, I present those of both system package managers, and application-specific package managers. However, because it is the latter which have the most structuring impact on programming language ecosystems, I will focus on them in more

[1]This is the case of yarn [309], which is an alternative to npm [256] to install npm packages, and of esy [231], which is a package manager for OCaml and ReasonML that supports both opam [107] and npm registries.

[2]For instance package registries internal to a company, or GitHub's own package registry [220] which supports a number of package managers.

[3]This is the case of the OCaml package index `https://opam.ocaml.org/packages/`, and the Coq package index `https://coq.inria.fr/opam/www/`. Both of them are based on the opam technology.

[4]This is the case of the Go programming language whose package manager expects URLs of git repositories. These repositories contain the required meta-data to install the package. Therefore, there is no central package registry but a package index can still be useful, and there exist several like GoDoc `https://godoc.org/`, and GoSearch `http://go-search.org/`.

[5]This is the case of the ReasonML "redex" package index `https://redex.github.io/`. The listed published packages are in the npm repository. The index also supports listing unpublished packages on GitHub.

[6]Such as the ones from the awesome lists movement `https://github.com/sindresorhus/awesome`

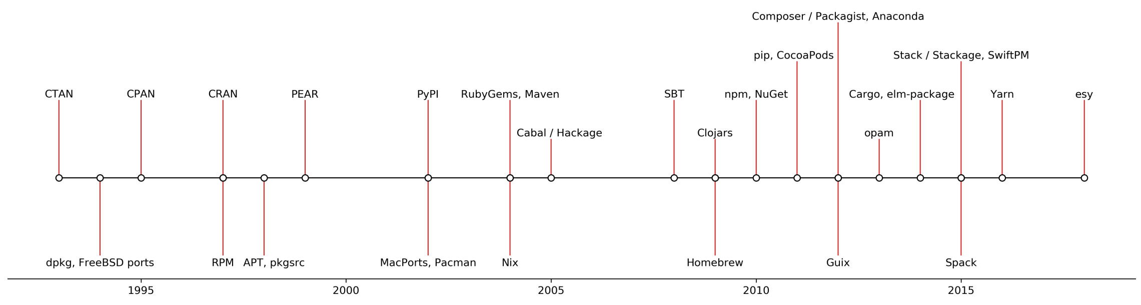


Figure 6.1: Timeline showing the creation dates of popular package managers. Application-specific package managers are on the top, system package managers are on the bottom.

details. Figure 6.1 gives the creation dates of a number of popular system and application-specific package managers.

6.3.1 System package managers

System package managers and registries started to be developed and used in the early nineties to solve the problem of reliable software installation on multiple workstations. One of the earliest academic publication on package management was presented at the USENIX LISA conference in 1990 [184].

Since then, package managers and their associated registries have become the core infrastructure allowing to create GNU/Linux distributions. Early and still very much used examples include dpkg [199] and the associated Debian package format (which is mostly used through higher-level tools like APT [302]), and RPM [90] (both a package format and a package manager, which is mostly used through frontends such as yum [291]). A lot of GNU/Linux distributions are based on just these two.

System package managers based on the model prevalent in the GNU/Linux world were also invented for macOS (for instance Homebrew [138]), and for Windows (for instance Chocolatey [242]). Nowadays, mobile phone operating systems like iOS and Android also have their package managers and registries (app stores), although they are used to distribute mostly non-free software, and they do not support inter-dependencies, contrarily to all the previously mentioned package managers.

Finally, Nix [83], and Spack [104] are ones of the few system package managers currently in use which are the product of academic research.

6.3.2 Application-specific package managers

The history of application-specific package managers and registries begins at almost the same time, but is much richer.

The first application-specific, comprehensive package registry was CTAN, the Comprehensive TeX Archive Network [124]. This project intended to solve the issue of TeX packages distribution, in a pre-web, internet era. Before its creation, there were multiple, independently-managed archives, available through different technologies (FTP and e-mail in particular), with different file structures, and which would all make available a set of existing, freely redistributable TeX packages, that none could claim to be comprehensive. CTAN introduced a standard file structure, and a mirroring protocol, so that it would be easy for anyone to download any TeX package from a geographically-close server.

CPAN, the Comprehensive Perl Archive Network, was created in 1995 on the CTAN model, and CRAN [137], the Comprehensive R Archive Network, was created in 1997, inspired by both CTAN and CPAN. Despite the name similarity, and the assumed heritage, each of these registries has grown independently and has developed their own practices.

Of the concerns and issues that were solved at this time, not many are still relevant today. In particular, modern package managers now assume that the original source code location will continue to be available and therefore do not necessarily mirror the source. With fast internet connection and CDNs (content delivery networks [42]), the geographical location of the server hosting the source code is no longer an issue either.

Many more package managers have been developed since then for various ecosystems (see Figure 6.1), and sometimes a single ecosystem got several competing package managers or registries. PEAR [222] was a package repository with a strong curation policy that was created in 1999 for the PHP ecosystem, but since 2012, it has mostly been replaced[7] with a new system (Composer / Packagist [208]). PyPI [234] was introduced in 2002, but pip [227], which is nowadays the official package manager to interface with the PyPI registry, was only created in 2011; and Anaconda [9], which is a package registry for scientific computing, has been introduced in 2012, and also contains a database of Python packages. Hackage [127] is used to distribute Haskell packages since 2005, but Stackage [264] is a popular alternative created in 2015 to bring more stability. npm [256] is the official Node.js package manager since 2010. Due to its versatile nature (since it allows uploading any type of binary package), it is used for more than just distributing Javascript. Yarn [309] is an alternative package manager to interact with the npm registry that was introduced in 2016. In some ecosystems (such as the Haskell ecosystem), Nix [83] has emerged as a strong alternative to using an application-specific package manager.

6.4 Socio-technical options for designers of package managers

A wide design space has already been explored by creators of package managers. However, there has been no survey describing the various options available, and their possible consequences. This is an attempt at starting such a categorization.

[7]Cf. this Stack Overflow answer: `https://stackoverflow.com/a/34200828/3335288`.

6.4.1 Options for package registries

There are a number of technical options to implement a package registry. Some will have consequences for the package manager (client); some will have social / cultural consequences on the way the ecosystem is organized. Finally, there are also some policy options that can be related to or independent of technical assumptions, and which can naturally have social / cultural consequences as well. Decan *et al.* [77] have argued that policies of package registries are important and can influence the overall ecosystem structure, and the behavior of maintainers. For a review of the cultural differences among different package ecosystems, see Bogart *et al.* [34].

What is uploaded to the registry? The registry can accept uploads containing only package meta-data, in which case the meta-data should contain a URL for the sources, from which the registry or the end-user (with the help of a package manager) may fetch them. Or, it can require package uploads to contain sources as well, or even pre-built binaries.

If the registry does not require uploading pre-built binaries, then the package meta-data typically contains build instructions that can be executed, either by the registry, or by the package manager.

Historic package repositories such as CTAN, CPAN, CRAN, Debian, require the uploading of the sources. This ensures that the registry can continue to serve them even if the official source location is down.

This is also the case of more modern registries like RubyGems. However, while Ruby packages generally contain only source code (Ruby being an interpreted language), nothing prevents anyone from including pre-built binaries in their packages.

Similarly, npm accepts packages containing any kind of content, either interpreted source code, or pre-built binaries.

Finally, package repositories like opam, MELPA, or nixpkgs only accept package meta-data to be uploaded.

What is distributed by the registry? The registry makes available at least some meta-data for each package, but it may, in addition, serve the sources of the package, and possibly a pre-built version. If the registry serves the sources but does not require uploading them, then it must mirror them. If the registry serves pre-built binaries but does not require uploading them, then it must be able to build them automatically.

For instance, opam serves the package meta-data and can optionally serve the sources [106], while nixpkgs serves package meta-data and provides a binary cache that can be used in substitution to install packages that have been built on the server. Finally Debian serves package meta-data, source code, and binaries.

Type of storage The traditional approach is to host an archive containing hierachically-organized files (either just package manifests, or even their source code). This is a technically simpler solution as it is easy to set up a new archive, to mirror an existing one, or to copy the entire content of the archive locally (especially when the archive only contains package manifests).

This archive can optionally be versioned as a whole, for instance as a git repository hosted on GitHub. In this case, the package manager can be configured to fetch the archive using git or the tarball automatically generated by GitHub.

For instance CocoaPods, homebrew, and nixpkgs use a git repository to store the content of the whole archive, and the archive can be fetched directly from GitHub.

Nowadays, there are many registries that rely on a database instead. This is the case in particular of npm, RubyGems, and PyPI.

How can the registry be browsed? In the case the registry is an archive, its content can be accessible through protocols like FTP or HTTP ("file-system-based registry"), or the full archive can be only downloaded as a large tarball, to be unpacked and browsed locally ("portable registry") [205].

A combination of both is also possible: for instance, an opam registry serves both individual package manifests and a tarball containing the content of the whole archive and used to synchronize a local copy.[8]

In the case the registry is a database, it is typically accessible through a RESTful API [94] (although other types of APIs like a GraphQL API [43] are also possible). This API may be documented or not depending on whether the authors of the registry view this API as a public interface that anyone can rely on, or as an entirely private interface to be used only between their registry and the package manager associated with it. However, stable and documented APIs are useful not just to build competing package managers (clients), but also to build services like Libraries.io [206] that present information about packages differently, and are much helpful to researchers (cf. Section 2.2.2).

It is possible for an archive-based registry to still provide an API. For instance, this is the case of CPAN [6].

When a registry is accessible through an API, it usually provides an endpoint to list all the packages available, but this endpoint may return incomplete results, in particular if there is a notion of private packages.

Access control to download a package Accessing the entire registry or a subset of packages may be restricted to properly authenticated and authorized individuals.

[8]This URL will download a single manifest file: `https://opam.ocaml.org/packages/0install/0install.2.14/opam`. This URL will download the full archive: `https://opam.ocaml.org/index.tar.gz`.

An enterprise registry, as supported by npm, would typically require authentication to access anything from the registry. Some registries also support publishing both public and private packages. This is the case of npm, and of GitHub's registry [220]. Authentication is only necessary to access private packages.

Technology to publish a new package Package upload can be handled by the same tool as package download (the package manager), or by some other tool. A solution that is very easy to implement is to create a GitHub repository of package meta-data (manifest files) and to allow uploading a package by creating a pull request. For users, it is likely however to be a more complex workflow than having a publish command in the package manager.[9] Some wrapper tool, like opam-publish [110], can be provided to automate as much of this workflow as possible.

An alternative solution adopted by many early package registries is to have a web-form to submit a new package.[10]

Access control / policy to publish a new package Most registries that accept publishing through a command will require the user setting up an account first, but registration will be open to anyone and effective immediately. Some, like CPAN,[11] have an account application process.

The registries that rely on pull requests to add or update packages usually have a peer-review process to ensure that the package meta-data meets some basic standards (but this does not mean that the content of the package itself is reviewed). They usually do not require the person publishing the package to the registry to have any connection with the package author (which on the other hand is assumed in registries relying on a publishing tool).

Note that the review process may not scale properly unless specific efforts are made to extend the reviewer team.[12] While the review process is likely to result in higher quality package, it may also incentivize package authors to put less effort to prepare their submission because they can rely on the review to catch problems [85].

Finally, some registries can have a peer-review process of the package content before they accept it. This is (or was, since no new package have been accepted since 2013) the case of PEAR, the first PHP package registry. Every package submission required its own RFC (Request For Comments) and ended up with a vote.

Technology to publish a new version In most cases, the technology for publishing a new version is the same as for publishing a new package, but there are some exceptions. For registries

[9]See some critics of this workflow at: `https://discuss.ocaml.org/t/opam-experiment-and-future-developer-experience-improvements/493`.

[10]This is the web form to submit a package to CTAN: `https://ctan.org/upload`. And this is the web form to submit a package to CRAN: `https://cran.r-project.org/`.

[11]This is the web form to apply for a CPAN account: `https://pause.perl.org/pause/query?ACTION=request_id`

[12]For instance, nixpkgs is implementing some changes to encourage more people to participate to the reviewing effort: `https://github.com/NixOS/rfcs/blob/138600b/rfcs/0039-unprivileged-maintainer-teams.md`.

where package submission is via a web-form, it can be the case that the maintainer of the package then gets granted a more direct access to the archive, where they can upload new versions without going through the form again. In some registries, each package gets its own version control repository, in which case new versions are published through this repository.

For instance, the conda-forge registry accepts submission of new packages through pull requests on a central repository (`https://github.com/conda-forge/staged-recipes`), but afterwards, a new GitHub repository is created on which the maintainer can upload new versions with or without peer-review.

Finally, new versions of a package can be automatically picked from its development repository. This is the case of MELPA: the initial submission is via a pull request on a GitHub repository, but then every new commit is considered to constitute a new version. MELPA Stable only considers the subset of tagged commits.

Access control / policy to publish a new version Most package registries which accept publishing via a tool will consider that publishing a new version can only be done by the user who published the first version, or other users that have been delegated permission. This is also the case for more traditional registries like CTAN, CPAN, CRAN, and Debian, and of conda-forge for which the initial submission is via a pull request.

It is interesting to note that while application-specific package registries allowing submission via a tool usually assume that the submission is by the author of the package, this is not the case of system package registries like Debian, Homebrew, or nixpkgs, nor of the application-specific registry conda-forge. Thus the person (or persons) that are allowed to publish new versions in the latter registries differ from the development team of the corresponding package.

The Elm package manager is special in that it does not have its own authentication system, and instead piggy-backs on GitHub's own access control. It considers that anyone with the permission of publishing a new tag on a GitHub repository is legitimate to publish a new version on the package registry.

For package registries that allow submission of a new version through a pull request, the new version will undergo the same kind of review as for a first version. It is usually not necessary that the submission of the new version is done by the user who submitted the first version.

Finally, it is worth noting that while some ecosystems value very short release cycles, others like R / CRAN do not, because of the heavy review process.[13]

Package naming Most package registries accept any name (in a restricted character set that typically includes alpha-numeric characters, dashes and / or underscores). However, more and

[13]On the CRAN policy page [63], we can read: "Submitting updates should be done responsibly and with respect for the volunteers' time. Once a package is established (which may take several rounds), "no more than every 1–2 months" seems appropriate." Decan *et al.* [77] have found that CRAN packages do receive significantly less updates than packages in other ecosystems. As they noted, this is probably a consequence of this policy.

more package registries allow (or enforce) scoping, which means that the name is composed of two parts (usually separated by a slash), the first one is the name of the owner (individual or team), and the second part is the name of the project.

For instance, npm supports scoped and unscoped package names (but only scoped names are authorized for private packages), while the Elm package manager and GitHub's package registry require scoped package names.

Removing a version or a package It is usually considered a bad practice to remove a version of a package after it has been published [209]. This was authorized nonetheless by several package managers like npm, until they decided to restrict this practice after the leftpad incident [304].

GitHub's registry similarly restricts any package or version removal (exceptions may be granted in special cases by contacting GitHub staff) [221].

When packages are submitted through pull requests on a repository, the maintainers of the repository will generally reject pull requests removing packages except in very specific circumstances. However, they may accept pull request fixing package manifest files, even though doing so without amending the version number would be considered bad practices in other ecosystems (such as Debian).

Some package registries, in particular for system package managers, prefer to keep a single version available, and in this case it is normal practice to remove older versions (this is just an update to the package and the version number is changed).

Finally, note that in registries striving to provide a consistent set of packages at a single version, like Debian, but also like CRAN or Stackage, packages may be removed from the set when they do not build with the available versions of their dependencies.

Tooling for collaboration and development The package registry may make some development tools available either optionally or automatically. Tools can include a version control system, a bug tracker, continuous integration, etc.

This was most useful before the rise of very large software forges like GitHub, which make it easy for anyone to have access to this kind of tooling today, and imposing every package to be developed in the same place is not likely to be successful.

However, when publishing a package to the registry is not done by the developers of the package (as in Debian, Homebrew, or opam), it can be useful to have a bug tracker to report issues specific to the package and not the underlying software.

Automated checks Linters are useful to avoid frequent mistakes. They are usually applied to the manifest file, either by the publishing tool, or via continuous integration on the package repository. It is generally helpful to be able to run the linter locally, and without publishing anything in case the linter is successful, so the linter should be made available as a separate command, even if it is run automatically by the publish command.

Continuous integration can also test the build of the package either before, or after it is submitted, and possibly notify maintainers of failing packages.

Version curation Some registries intend to provide a curated set of packages. This includes registries for system package managers, e.g. Debian, but also application-specific registries like Stackage, which is a curated version of Hackage, the Haskell package registry.

Only providing a curated registry in a given ecosystem may lead to frustration from some users wanting to rely on older releases. This can lead to initiatives such as Microsoft's MRAN (`https://mran.microsoft.com/`), which provides daily snapshots of past CRAN versions.

Delay to release A lot of application-specific registries, and some registries for system package managers, make packages available very soon after they have been submitted: this is the rolling release model.

On the other hand, curated registries are more likely to have a release cycle with release events when the whole public registry is updated at once. For the users who want stability, this is usually a great solution, but for some other users who want to take advantage of new versions as soon as they are available, it can be frustrating if the main registry in the ecosystem is not on a rolling release model. This is for instance the case of the R ecosystem (CRAN is curated and new versions of packages are only distributed with new releases of the R language), and it leads maintainers of actively developed packages to host their own registries [33].

6.4.2 Options for manifest files

The manifest is the file that is used to store package meta-data that the package registry and package manager will use. Some of this meta-data is purely informational and aimed toward humans (author names, package description), some is informational but may be automatically parsed by tools (license information stored in SPDX format [260] may then be used to inform package authors / users of license compatibility issues), some can be used to implement package search and filters (tags), and some is mostly intended to be read by tools such as the package manager (dependency information).

File format The manifest file format can be based on a standard DSL syntax (JSON [64], YAML [25], TOML [233], S-expressions [246]...), can use a custom DSL syntax, or can use a normal programming language (typically the ecosystem's programming language).

The last solution is common with interpreted languages (Perl, Ruby, Python...), and its main advantage is that no specific parser is needed. However, it also means that interpreting the manifest can lead to arbitrary computation, which is not likely to be viewed as a good thing.

The main advantage of using a standard DSL syntax is that it will make it easier for any external service to parse the manifests (this is the reason given by CocoaPods to move from Ruby to JSON [85]).

Nix uses its own language for package management, which is a non-Turing-complete language where some computations are possible (including if-then-else expressions, and string, list, set concatenation / merging).

Version of the package Usually the version of the package will be in a constrained format usually limited to numbers separated by dots with possible trailing alpha-numeric qualifiers (e.g. beta1). Some package managers / registries enforce the use of a format matching the Semantic Versioning definition [232]. In general, they cannot however, verify that Semantic Versioning is actually respected (no breaking change in a minor release, no new feature in a patch-level release). An exception is the Elm package manager which automatically bumps the version number based on the API changes. Some registries use alternative versioning policy: for instance Hackage uses the Haskell Versioning Policy [44] (that was drafted before Semantic Versioning was introduced).

Besides the meaning that the version number conveys to humans, one of the most important aspects of version numbering is the ability to sort version numbers. Unfortunately, there is no unique convention regarding version number sorting. Semantic Versioning defines that pre-releases are denoted by appending a dash followed by some pre-release identifiers, and that such pre-release versions come *before* the corresponding normal version. Debian's convention, also used by opam, denotes pre-release with a tilde instead of a dash [144, Section 5.6.12]. Nix's unspecified implementation[14] only gives precedence to versions with a trailing `pre` (possibly after a dot or dash separator) over normal version numbers.

Versions of the dependencies A manifest usually contains dependency information with version constraints. There are some exceptions to this: in registries providing a single version for each package, manifest files may be lacking any version information regarding their dependencies.

Most manifest formats accept both specifying depending on version ranges or exact versions of a dependency, and it is then a matter of convention what kind of range to use. It is usually advised for library authors to specify version ranges (as large as possible) to impose fewer constraints on depending application / libraries, while it is advised for application authors to pin their dependencies (specify exact versions) to maximize reproducibility.[15]

In some ecosystems it is common practice to use open ranges (ranges whose upper bound is larger than the last published version). This will typically be the case in ecosystems were library authors are expected to follow semantic versioning (in which case the upper bound will be the

[14]See: `https://github.com/NixOS/nix/blob/b095c06/src/libexpr/names.cc`.

[15]See "Why do binaries have Cargo.lock in version control, but not libraries?" in the Cargo Book's FAQ [252, Chapter 5].

next major version, and it will be a strict upper bound). In some ecosystems, it is frequent that packages do not specify any upper bounds. This is the case for instance in the opam registry, but manifests can be amended later on to add upper-bounds when a compatibility breakage with a newer version of a dependency is noticed.

6.4.3 Options for package indexes

The design of package indexes is important because it may influence the discoverability of packages, and which dependencies software developers will rely on in the end.

Package sorting Some package indexes only provide a search engine but no list of all packages. This is the case of npm in particular.

Among the package indexes that provide a list of all packages, the most standard sorting criterion is package names (Debian, MELPA, nixpkgs, opam, RubyGems...). Alternative criterion can be based on initial publication date, date of last release, popularity indexes, etc.

The Elm package index has an unusual sorting algorithm that puts forward the packages by authors that are most active in the community (counted by number of talks given to Elm conferences), and resorts to alphabetic ordering for the rest.[16]

npm's package search supports sorting using several metrics (including popularity, quality, and maintenance indexes).

Package search and filtering mechanisms Most package indexes have a search mechanism using the textual information present in the package title, description, and possibly documentation. In some cases, this search is reduced to a simple filtering mechanism looking for an exact match of the searched string with a part of the package name.

Some more advanced searching tool exists, like Hoogle (`https://hoogle.haskell.org/`) which searches function by type signatures in all the packages in Stackage.

Some package indexes also allow filtering by tags (freely defined by the package maintainer), or categories (chosen from a predetermined list).

Nixpkgs filters out packages with non-free licenses by default. Some more advanced filtering mechanisms based on license types could be designed, although I do not know any package index having implemented them.

Metrics to display Given that most registries accept package submission without any quality assessment, there can be a number of available packages designed to solve the same problem, and it is a challenge to present information that will enable the user to choose. Some package indexes display a number of metrics to allow discriminating between various packages. These metrics can include popularity metrics based on number of GitHub stars, number of forks, number of

[16]`https://github.com/elm/package.elm-lang.org/blob/a8b9f08/src/backend/Memory.hs#L223-L324`

downloads, number of other packages depending on it, maintenance metrics based on the date of the last release, the date of the last commit on the source repository, the number of open issues, etc. Some indexes show, for each package, the exact list of its dependencies and its reverse dependencies.

Embedded documentation To help users learn about a package before deciding to use it, some package indexes display documentation extracted from the package. For instance, npm and PyPI display the package's README.

Some package indexes can rely on a unified documentation format to display the full package documentation. This is the case, in particular, of the Elm package index (the presence of documentation is even enforced when uploading a package to the registry).

6.4.4 Options for package managers

Package managers will be the tools that developers will directly rely on and interact with every day. Their design can have a strong impact on the productivity of developers.

Global, user-local, mutli-profile, or project-local installation Naturally, system package managers, such as the one provided by any Linux distribution, support installing packages globally (i.e. any user will then be able to run the installed software). Some system package managers additionally provide the ability to install a package locally in a user's profile, without requiring administrator permissions.

Application-specific package managers can provide the possibility of installing packages globally as well, although this may create conflicts with other package managers installing in the same global location. On the contrary, application-specific package managers installing packages in a local user profile will usually do so in their own directory, and add this directory to the user's `PATH`.

Some package managers (opam for instance) support installing multiple package sets in distinct profiles that can be switched to, allowing parallel installation of multiple versions.

Finally, many package managers provide a way of installing a package set specific to a given project. In some cases, this is managed by a tool external to the package manager itself (for instance venv in Python). This is especially useful so that developers can easily and reliably set up an environment containing all the packages they need, to work on a software project, regardless of what else they are working on at the same time / have installed on their system.

Multi-registry support Most package managers support multiple registries although there is often a specific registry that is considered to be the central hub for the ecosystem, and that is configured by default in the package manager.

Alternative registries are useful to distribute private / proprietary packages, development versions of packages, to distribute packages that do not meet the policy requirements of the central package registry, or to reuse an existing package manager for another ecosystem.

Support for installing from source repositories Some package managers support installing packages directly from the source repository (typically a GitHub repository) instead of installing from a registry. This is often a way to access the development version of a package, without any effort from the package maintainer, other than maintaining a manifest file within the sources.

Some package managers, like the Go package manager, only support this method, which means that there is no need to manage a package registry.

Local binary cache When packages can be installed locally to a project, it can frequently happen that the same package will be needed for multiple projects. This may lead to building it and storing it multiple time, which is a loss of time and disk space.

A solution around this problem is to build and store packages only once, in a directory managed by the package manager, and to use symbolic links, or any other virtual installation method to ensure that the required packages are made available in the environment local to a project.

A good model to implement this kind of local binary cache is the store that the Nix package manager uses. This is the model that has been followed to implement the new build system in the Cabal package manager [45, Chapter 5 "Nix-style Local Builds"].

Management of diamond dependencies When two direct dependencies share a common indirect dependency, there are two ways of proceeding: either install two copies of the indirect dependency, or find a version that is suitable for both.

The first solution is the one used by npm, and it makes the dependency selection algorithm much simpler, at the price of disk space, and of causing trouble if what this common dependency ends up leaking in the main program at two different, and incompatible versions. For some programming languages, such solution will simply be impossible to implement, because of namespace conflicts.

The second solution requires solving an NP-complete problem, and in some situation the result may just be that two packages cannot be installed together [183].

Management of dependency updates The package manager may provide some help to discover direct dependency updates, evaluate whether the new version is compatible, or worth to update to (for instance by showing release notes, or even an API diff like the Elm package manager does), whether updating is possible given the constraints imposed by the other packages, and finally update the manifest file and the installed set of packages is such a decision is taken.

Build system integration In some programming language ecosystems, the package manager and the build system are tightly integrated or even part of the same tool. This usually requires a high level of control on the ecosystem, and thus mainly happens when the designer of the package manager is the designer of the programming language, and the programming language is relatively recent. While such tight integration can help the build system be aware of where the libraries are located, it is also important to leave open other scenarios where the build system is used without the package manager. Otherwise, it may seriously harm any effort by other package manager / registry maintainers to distribute a piece of software written in this programming language.

6.5 Package distribution in the Coq ecosystem

6.5.1 Historical overview

The Coq development team has provided a package distribution method for more than twenty years.

Coq 6.1, released in 1997, came with forty so-called “user contributions” or “contribs” for short, that were distributed in a single tarball,[17] and were organized in sub-directories corresponding to the geographical location of the authors.

Coq 6.3, released in 1999, came with over fifty contribs.[18] These contributions could be all downloaded as a single tarball, or as individual tarballs. An index, with a search engine, was provided, and each package had its own presentation page embedding a description, the authors' names, and the package's README, that was automatically generated from a “description” file. As with CTAN, this was only a registry and an index, but no support was provided for locally fetching and installing a given package (beyond the installation target in the included Makefile).

The initial submission process was very basic: the author of the package had to complete a text-form (i.e. create the package manifest file), and then send it by e-mail with a link to where the package could be downloaded. Later on a web-form was introduced.[19] After the package had been submitted, it would be maintained by the development team to stay compatible with new releases of Coq (which frequently contain breaking changes). This model corresponded to a mental model where proofs / formalization works, once achieved, are not meant to evolve beyond what is needed to stay compatible with the evolution of the underlying proof language. It was particularly well suited for packages that would typically be submitted by PhD students, ready to share their work and move on to other things.

But this model also had drawbacks: it could not scale to hundreds, or even thousands of packages, because the manpower to do the maintenance work was limited; there was no quality

[17]This tarball can still be downloaded from the Coq website: `https://coq.inria.fr/distrib/V6.1/contrib/`.
[18]See `https://web.archive.org/web/20001029020059fw_/http://coq.inria.fr/contribs1-eng.html`.
[19]See `https://web.archive.org/web/20140813011527/http://coq.inria.fr/pylons/contribs/new`.

assessment, and some users complained about this; it was not suited for libraries whose authors kept maintaining and evolving, and these started to be distributed outside of the "Coq contribs" scope.[20]

In October 2013, so not long after opam (the OCaml package manager) [107] was first released, the idea of turning it into a package manager for Coq was mentioned publicly for the first time, by Thomas Braibant.[21] This idea was all the more reasonable given that Coq itself is written in OCaml, and many packages contain plugins, i.e. OCaml files interfacing with the Coq API. In fact, the question of the distribution of Coq packages had been recently explored by several people (including Guillaume Claret and Cyril Cohen [53]), and quickly gained support from a number of other developers and users. Over the following year, the Coq package registry was created by Guillaume Claret, Guillaume Melquiond, Enrico Tassi, and others, and started to be advertised in November 2014 (post on Guillaume Claret's blog [52]), and more and more over time as the preferred method for installing Coq packages.

The package manifests are maintained in a specific GitHub repository (`https://github.com/coq/opam-coq-archive`), which is published as an opam registry; and instructions are provided on how to use it with opam (which requires installing Coq itself with opam). On the other hand, the Coq package index (`https://coq.inria.fr/opam/www/`) uses a custom website infrastructure and design, that is quite different from the OCaml package index (`https://opam.ocaml.org/packages/`). By convention, all Coq package names are prefixed with "coq-" to avoid any clash with packages hosted in the official OCaml registry.

6.5.2 Open issues, design exploration, future work

The current method to distribute Coq packages still suffers from a number of issues.

To start with, installing Coq with opam is non-trivial: Windows's support is limited;[22] CoqIDE requires dependencies (GTK) outside of the scope of opam, and installing them may be more or less well documented depending on the system / Linux distribution. It is possible to install Coq using a system package manager, or an installer for Windows or macOS, but these installation procedures do not make it possible to use Coq packages installed through opam.

Mastering opam (even just as a user) requires significant efforts that some Coq users, like those with a mathematical background, rather than a computer science / technology background, are not necessarily able or willing to undertake from the start.

Furthermore, using opam requires compiling everything from source, including the OCaml compiler, the Coq software, and the requested packages, which may add up to dozens of minutes of build time (when this is not hours), given how long some Coq packages take to build. Related

[20]See `https://sympa.inria.fr/sympa/arc/coq-club/2013-10/msg00119.html`.

[21]See `https://sympa.inria.fr/sympa/arc/coq-club/2013-10/msg00096.html`.

[22]Windows is not even mentioned in the opam install page (`https://opam.ocaml.org/doc/Install.html`), nor in the FAQ. It is only mentioned in the unofficial packages at the bottom of the Distribution page (`https://opam.ocaml.org/doc/Distribution.html`).

to that is the fact that opam does not yet have any local binary cache support, which means that installing a package set for a specific project would require recompiling everything that is needed from scratch, just for this project.

Finally, the Coq package index needs some improvements to be easier to navigate, and to put forward a curated list of most recommended packages.

With Michael Soegtrop and Enrico Tassi, in particular, we have explored the idea of creating a Coq platform, which would allow to easily install a curated subset of Coq packages. This has been first experimented in the Windows installer, which now includes a set of commonly used plugins and libraries. The idea, now largely supported, is to define a clear inclusion and version selection policy for the Coq platform, and to provide multiple ways of installing it (a Windows installer, a macOS installer, but also a Snap [47] package, an opam package, and to encourage maintainers of Coq packages on other registries to provide a Coq platform package as well). This platform will be mainly targeted at newcomers, users with less technological experience, and users who want a stable and curated set of packages they can rely on. It will likely be maintained closely to, but independently of Coq itself, and will be released with a short delay after each new Coq release.

The model of such a platform already exists in other ecosystems with the Haskell platform [129], the Scala platform [255], the TeX Live distribution [238], and is being explored by others, e.g. the OCaml plaform [212]. This is an alternative model of package distribution that is complementary to the package manager / registry model (although platforms often include a package manager so that users can install additional packages when they need them), and that shares some similarities with registries with release cycles like CRAN.

We have also explored solutions to make expert users more productive. One question in particular was how to distribute an up-to-date, pre-built, development version of Coq that users can use in CI. Erik Martin-Dorel has proposed a solution based on pre-built Docker [143] images, and I have proposed a solution based on Nix and Cachix [84], a tool allowing to create a custom public binary cache for Nix. Both of these solutions have been implemented to test packages from coq-community (see Chapter 7).

Following my explorations around Nix, several Coq developers are now considering making the Coq package index support two package management technology at once: opam, which will continue to provide more flexibility regarding the combination of versions to use, and Nix, which will come with a binary cache, and will provide a fast and reliable install experience, and other advantages such as sandboxed environments.

6.6 Conclusion

Package ecosystems can play a major role in a programming language success, and the way packages are distributed (in particular the policies of package registries) can strongly impact the shape and structure of a package ecosystem. Because no one had yet (to my knowledge) published any survey on socio-technical options for the design of package managers, registries, and indexes, I have started one by listing many such options, and giving real examples of their application in popular package managers. The community of package manager designers is much more a practitioner community than an academic community, but they are self-organizing so that ideas can be shared across ecosystem borders (through initiatives such as the Manifest podcast `https://manifest.fm/`, by Alex Pounds and Andrew Nesbitt, and `http://package.community/`, started by former npm maintainer Kat Marchán). Such a survey would likely be of interest to them.

In the second part of this chapter, I focused on package distribution in the Coq ecosystem: it has a long history, and it evolved by successive adaptations to the issues that developers and users encountered. No one had yet written a complete summary of this evolution. I also presented recent progress and design exploration to which I participated, and which intended to address recently expressed needs of new users and project maintainers.

CHAPTER 7

PACKAGE MAINTENANCE

7.1 Introduction

Rich application-specific package ecosystems are most often made of open source libraries which are contributed by volunteers. But these libraries must also be maintained to accommodate the evolution of the technology on which they are based. The programming language itself may be evolving at a slow or fast rate; the other libraries upon which a given library depends can receive compatibility breaking changes even if the language has not, and it may be important to stay compatible with the latest version for convenience or for security; the platforms to support (e.g. web browsers, or processor architectures) are likely to evolve, but depending on the abstractions upon which a library is built, it may matter or not; the state-of-the-art algorithms may have become better; etc. Overall no software can completely avoid the need to be maintained, and software libraries, even small ones, are no exception,[1] even though the amount of maintenance they require may be very small.

Commercial library vendors will generally sell support, or new licenses for up-to-date versions, but in the case of open source libraries, in particular small convenience libraries, their author usually did not decide, when they created them, that they would maintain them forever. While

[1]At the current time, there is not much evidence for this claim. Based on empirical studies, laws of software evolution have been devised [175], which state among other things that software systems must be continuously adapted, or they end up being less and less useful. However, these laws were carefully restricted to so-called "E-type software systems", which are software that require evolution because the world, and the expectation of users, evolve. These E-type systems are opposed to P-type systems, which are based on heuristics which can be refined further and further, and S-type systems which are programs that are proven to behave exactly as their specification mandates [174]. Nonetheless, experience in the Coq ecosystem, and more generally in the formal method community, would tend to show that these S-type systems are no exception to the general rule that software must be maintained, and even proofs, when formalized on computer, have to be maintained, despite their mathematical immutability.

creating and sharing a library may have not cost much to the author (or even benefited them in the form of personal satisfaction, external contributions, or new, innovative and useful libraries depending on theirs), the cost of maintenance can be much higher,[2] and does not always come with any incentive (as the author of the library may not even use it anymore). On the other hand, users that are currently using the library have much more incentive to contribute to maintaining it, but this is not always as easy for them as it is for the author.[3]

The question of who should be responsible for maintaining an open source library is therefore far from trivial. And the answer may actually depend a lot on the way the ecosystem is structured (cf. Chapter 6).

In this chapter, I focus on the issue of packages that are maintained by a single person (usually their original author), while being depended on. I argue that these packages are more likely to become unmaintained, and pose a specific threat on the health of the ecosystem. In a first section, I estimate the prevalence of such packages, and I find that they are numerous. Then, I present mitigation models for users of these packages, in particular the possibility of creating a hard fork. Finally, I present a model of community organization, which emerged in several ecosystems, that can simplify the process of hard forking, and provide a new home for unmaintained packages. I have created such an organization for the Coq ecosystem over a year ago, and it has been already quite successful.

7.2 Estimating the prevalence of single-maintainer packages

The advance of reuse facilities, such as package managers and registries (cf. Chapter 6), has led software projects to depend on more and more, smaller and smaller, utility libraries, instead of relying only on widely-known and trusted ones.

But all software libraries must be maintained, and the main issue of small libraries is generally that they have fewer maintainers, most often a single maintainer, who might become less active or outright missing for a number of reasons. While the extreme case of trivial packages [2] can be trivially solved by copy-pasting the code, and maintaining it within a larger source code base, I am interested in the frequent case of non-trivial libraries that are worth keeping as separate dependencies (to avoid duplicating innovation and maintenance work in the various projects depending on it) but are still small enough to have a single maintainer.

I propose the following criterion to define the kind of libraries I am talking about: an open source library which requires by itself several hours of maintenance each year, which is depended

[2]Maintenance costs are frequently cited to be at least as high as development costs in the total lifecycle of software [178]. However, these conclusions come from studying large systems in the software industry and do not necessarily apply to the (possibly very small) open source libraries that are discussed here.

[3]Difficulties can range from lack of knowledge of the source code, to not having access to the credentials required to upload new versions to the official distribution channels, to not being able to mobilize as well the other contributors. A solution to the second and third problems can be for the author to trust one motivated contributor, and give them access to the required credentials, but misplaced trust can have dire consequences, as has been recently demonstrated by the event-stream incident [19, 82].

upon directly by at least two actively maintained software projects by distinct developers or developer teams, and which is maintained by a single developer (i.e. a single developer has push-access, even if changes might be contributed by others through pull requests or e-mailed patches). I conjecture that such libraries are quite frequent in many package ecosystems. Given that it can happen that maintainers of such libraries stop responding to requests and updating their library for extended periods of time, this type of packages represents a very specific threat to the health of ecosystems.

To validate this first conjecture, I use the dataset provided by Libraries.io [206] (cf. the corresponding Jupyter notebook [321]). This dataset includes a table of packages associated with repositories, containing more than 3,000,000 packages, including more than 1,000,000 Go packages, 925,000 npm packages, 250,000 PHP packages (Packagist), 200,000 Maven packages, 170,000 Python packages (PyPI), and 150,000 Ruby gems. I first restrict this dataset to the 2,500,000 packages whose repository is located on GitHub, because the dataset contains more complete data for these.

A first filtering step of packages with more than one dependency, and a repository size of at least 10 KB (in an attempt to filter out trivial packages), makes this number go down to 257,000 packages, including 116,000 npm packages, 29,000 PHP packages, 22,000 Ruby gems, 19,000 Python packages, 18,000 Maven packages, and just 9,500 Go packages. In most ecosystems, this first filtering step has cut down the number of packages by a factor of ten, but in the case of Go, the factor is a hundred. This can be explained by the absence of package registry for Go, which forces Libraries.io to look for Go packages directly on GitHub instead, where it finds many projects that are not meant to be reused as libraries.

This first filtering step is not sufficient because I want to find only packages that are used in actively maintained projects from different owners (individual developers or teams). Therefore, I use the repository dataset provided by Libraries.io, which contains about 34,000,000 repositories, and I extract those that were pushed to in the last six months before the Libraries.io dataset was published. The number of remaining repositories is about 800,000 (once again this involuntarily excludes repositories outside of GitHub, for which the dataset does not contain the last pushed date).

Finally, I join the package and repository data by relying on the dependency dataset provided by Libraries.io, which includes about 390,000,000 dependencies. This finally restricts the set of packages that are depended upon by two actively maintained projects from distinct owners to about 65,000.

Sometimes, multiple packages are maintained in the same repository (more than 1,500 packages in the DefinitelyTyped repository,[4] although this is a very special case, the next biggest monolithic repository being the Babel repository with 286 packages). In this case, it is hard to estimate different maintenance indexes for the different packages, and giving the same

[4]DefinitelyTyped is a community repository gathering TypeScript type definitions for otherwise untyped Javascript packages: `http://definitelytyped.org/`.

maintenance index to thousands of packages would completely bias the statistics, so I decide to keep only one package per repository (the package that is most depended upon).

Out of the remaining 50,000 packages, 18% have just a single contributor (which is worse than having a single maintainer). Next, I estimate the packages with a single maintainer.

For each package owner, I look on GitHub (using the GraphQL API which makes it possible to bundle queries about 40 owners in a single HTTP request) whether it is an organization, and how many public members it has. I found that about 33% of these 50,000 popular packages belong to organizations with at least two public members. Belonging to an organization does not guarantee that the package will keep being maintained, but it should prevent against a maintainer disappearing without notice, and no one having access to the repository.[5]

For the rest of the package owners, I approximate the number of maintainers by querying for the number of assignable users. Assignable users are a super-set of collaborators with write-access:[6] they correspond to organization members with read-access (when the owner is an organization[7]), and collaborators that were manually added on the repository itself [111].

Of the remaining packages that were not part of an organization with two public members, I estimated that only 33% have two or more collaborators. This leaves a large proportion of packages at risk.

7.2.1 Threats to validity

The goal of this section was to estimate the proportion of packages that are sufficiently popular to be used in two maintained projects by different owners, and yet have a single maintainer. Two kinds of errors could have affected the results: errors in the computation of popular packages, and errors in the computation of single-maintainer packages.

Popular packages could have been missed, and in fact it is certain that this was the case, since some popular packages are hosted outside of GitHub. Another possible error could have been with the maintenance criterion. Some projects should be considered maintained even if they were not pushed to during a six-month period (some feature-complete packages may need less frequent updates [285], especially in a slowly evolving ecosystem). Furthermore, some projects may have been pushed to after Libraries.io last refreshed the GitHub meta-data about them, and thus the dataset may have contained outdated information that could have resulted in marking these projects as unmaintained. Therefore, the number of maintained packages depending on a given package could have been underestimated, and the package excluded because of this. However, the resulting sample of popular packages is still rather large (50,000 packages). Bias in this sample

[5]To be fair, even if a package belongs to an organization with two or more public members, it could have a single administrator, and if credentials are needed to publish a new version of the package, then it is possible that the main maintainer did not share these credentials with anyone else.

[6]For privacy reasons, GitHub does not share the list of collaborators, or people with write-access on a repository, but it does share the list of assignable users nonetheless.

[7]This can also be used as a way of approximating organization membership, including non-public members.

could result mostly from the exclusion of non-GitHub projects, and of many packages hosted in monolithic repositories.

I computed dependencies between packages irrespective of package version numbers, so this could also lead to considering former dependencies that were dropped or replaced. Fortunately, the Libraries.io dataset contains more specific information regarding which version depends on what, so I plan to use it to refine this analysis in the future.

Then, the criterion that was used to approximate the number of maintainers is bound to have introduced some errors as well. However, I believe that it is more likely to have resulted in overestimating the number of maintainers of a package, rather than underestimating it. Indeed, it is not because someone can be assigned issues in a repository that they have access to all the required tools and credentials to be considered an actual maintainer of the package. Even if they are an actual maintainer, and are ready to take over from the previous maintainer for a while, if they do not have admin-access, they may not be able to nominate new maintainers, and this can harm the maintenance of the package after some time, in particular if they want to step down eventually.

Overall, I believe that these threats do not put any doubt on the main conclusion of my analysis which is that many popular packages have a single maintainer. However, I have not yet tried to evaluate separately for each ecosystem if the proportion of popular packages at risk was similar. I plan to do this as future work, as this would increase external validity of the results, and would avoid having larger ecosystems bias the overall results.

7.2.2 Related work

7.2.2.1 Project maintainers and developers

Yamashita *et al.* [308] analyzed the proportion of core developers in 2,496 GitHub projects. They used various criteria to define core developers, one of them using GitHub's collaborator API endpoint (through the GHTorrent dataset [121]). Unfortunately, access to this API has been restricted since then, and GHTorrent does not include this data anymore.[8] This is why I used the assignable users information instead.

My analysis is different from studies computing metrics such as *bus factor* (also called *truck factor*) [93, 276], and more generally estimating turnover risks [203, 243] within a software project, because I am focusing on the problem of who is able to integrate changes and publish new versions of a package. It can happen that popular packages receiving pull requests from many contributors still have a single maintainer that suddenly disappears. On the other hand, a project that is not subject to such risks because it belongs to an active organization, or has several maintainers, can still be subject to knowledge loss risks. Therefore, the two types of

[8]Cf. the commit in GHTorrent removing the support for this API endpoint: `https://github.com/gousiosg/github-mirror/commit/79c81bf`.

studies are complementary, and highlight different, but related, health risks that can affect a package ecosystem.

Avelino *et al.* [18] have found a significant proportion of projects having faced the event of a “truck-factor” developer stepping down, despite having only analyzed the 500 most starred repositories in the six most popular languages on GitHub. Less than half of the projects survived, generally because a new, or preexisting contributor took over. They interviewed these contributors that helped projects survive, and identified the difficulty of getting access to the repository as a significant barrier (when project maintainers had become unresponsive).

7.2.2.2 Unmaintained projects

Several studies have highlighted the fact that many open source projects are dormant or abandoned [157, 165]. However, this result would not surprise any GitHub user: many projects are personal projects or just never take off. A project threatening to become dormant does not pose the same risk to open source ecosystems depending on whether it has a lot of users, very few, or none beyond its author.

Valiev *et al.* [285] have found clear evidence that packages that have been able to gather a large community of users over time, characterized by a high number of reverse dependencies, are much less likely to become dormant. This expected result does not contradict the observation that there are a number of such packages that have a single maintainer and risk becoming dormant, despite having a community of users ready to help (even though this community does indeed reduce the risk).

7.2.2.3 Ecosystem health

Measuring ecosystem health is an important question that researchers have studied in the past [148]. It is still a highly active research domain, as witnessed by the currently running SECOHealth project [192]. My work relates to this literature by highlighting a health risk factor (single-maintainer projects) that could be integrated in ecosystem health assessment frameworks.

7.3 Mitigation models

When a package becomes unmaintained (the maintainer does not respond anymore to issues and pull requests, and does not push new commits or versions), what are the options facing the projects using it?

7.3.1 Removing or replacing the dependency

Upon the realization that a project depends on an unhealthy dependency, it can be time to reevaluate the usefulness of such dependency. Sometimes, the functionality brought by the dependency is not that useful, or an alternative, healthier package could be used instead. Removing or replacing the dependency can, in such case, turn out to be beneficial, although the migration can bear significant costs to the project, not necessarily at the best of time.

Furthermore, this is a solution that each project has to evaluate on their own, and while it might be feasible for some projects to drop the dependency, it might be significantly harder for others. Having many projects migrate away from an unhealthy library can further reduce its chances of surviving, thus threaten other projects for which migrating is too costly (for instance, because they are barely maintained themselves).

Previous work has explored mining data from projects having performed library migrations to automatically suggest candidate libraries to migrate to [272], and to automatically identify mapping between methods of the two libraries (in the context of Java projects) [8, 273].

7.3.2 Vendoring

Vendoring a dependency is the process of copying its sources within the project's sources, and building the whole thing, instead of installing the dependency with a package manager first, and building the project by relying on the installed dependency.

Depending on the language or build system support, vendoring dependencies might be trivial or not, but should always be feasible (at the price of renaming modules if proper scoping cannot be achieved otherwise, and adapting the build configuration). Go is well-known for having good vendoring support [132]. This is also the case of OCaml when both the vendoring and the vendored projects use the Dune build system [81]. Work is in progress (by Emilio Gallego Arias) to add Coq support to the Dune build system, including the compositional build feature, that makes it possible to vendor dependencies.[9]

Vendoring allows a project maintainer to integrate patches before they are integrated upstream. Some of these patches might have been found in unmerged pull requests from external contributors. However, this solution cannot be a long-term solution because it is more work for everyone to have to integrate patches manually, and at some point new pull requests cannot continue to be based on an unchanged base branch.

7.3.3 Forking

7.3.3.1 Definition

Forking has a broad meaning today in the open source world.

Early academic works which studied forking considered only what is usually denoted today as *hard forks*. For instance, in their 2012 paper [247], Robles and González-Barahona give a definition of a fork that includes requirements such as having a new project name, and a

[9]Cf. `https://github.com/ocaml/dune/pull/2053`.

disjoint community. The Hacker's Dictionary [240] even specifies that the two code bases must be developed in parallel, and have irreconcilable differences between them.

Some papers were dedicated to studying a single fork event, such as the LibreOffice fork from the original OpenOffice project after the Oracle acquisition of Sun Microsystems [103]. The "right to fork" has been discussed in works such as [300, page 64] as an essential freedom of free software. Nyman and Lindman [210] claim that forking is the most important tool to guarantee sustainability in open source development, and that the right to fork has a major effect on governance, even in the absence of any forks.

With the rise of GitHub, forking has taken a new meaning. *Development forks* [95, Chapter 8] are copies of the sources where a contributor makes changes to the code in order to submit them for review through the pull request mechanism.

But even before GitHub, forking was much more common and much less definitive that hackers and researchers alike seemed to believe. Nyman and Mikkonen [211] observed the presence of many forks on SourceForge, including forks claimed to be temporary and hoping to get their changes integrated upstream (that can therefore be classified as development forks). They also noticed the phenomenon of forking a project because it seemed to be abandoned, and not because of some disagreement. While this should still be denoted as a hard fork, there is only one project under development after the fork, contrary to the definition of the Hacker's Dictionary [240]. This is the kind of forks that we are interested in, in this section. We denote this sub-type of hard forks as *friendly forks*.

It should be noted that while Nyman and Mikkonen observed a number of forks of unmaintained projects, SourceForge had, at the same time, a takeover procedure to allow a user to take control of an abandoned project.[10] This may have reduced the need to create friendly forks, despite the fact that takeovers often happened without the consent of the project's previous maintainers (in case they were unresponsive).

[10]The procedure, mentioned by Khondhu *et al.* [165], was known as "Abandoned Project Takeover". It was advertised as an existing policy by Ross Turk, a SourceForge community manager, in an e-mail thread from 2007 (`http://lists.tlug.jp/ML/0710/msg00030.html`): "SourceForge.net does have a system for taking over a project. We call it our Abandoned Project Takeover system, and to activate it you just need to attempt to register a new project with the same name as the old one. The old admins will be emailed, and if they don't respond within a certain period of time, the project will be transferred to the new requester." It was also explicitly presented on the SourceForge website in 2010 (`https://web.archive.org/web/20100609115535/http://sourceforge.net/apps/trac/sourceforge/wiki/Abandoned%20Project%20Takeovers`): "We give the existing project administrators 90 (ninety) days to respond to these requests. If after 90 days there is no reply to our take over request notification, we will assign the project to the requester." This page was changed in 2011 (`https://web.archive.org/web/20110326034942/http://sourceforge.net/apps/trac/sourceforge/wiki/Abandoned%20Project%20Takeovers`): "If the project administrators do not respond, or you do not wish to contact them, you may register a new project under a new name on SourceForge.net and manually fork (duplicate) the source code and file releases as you see fit to continue its development." But despite this update that seemed to indicate that it was not possible anymore to take over an abandoned project without the previous maintainers' permission, such requests were still processed several years after: `https://sourceforge.net/p/forge/site-support/1636/` (2012), `https://sourceforge.net/p/forge/site-support/2384/` (2013, mentions that SourceForge's legal team is working at updating the process, but grants the request anyway), `https://sourceforge.net/p/forge/site-support/6629/` (2014).

7.3.3.2 Socio-technical issues when forking a package

When to advertise a friendly fork? While it is easy for anyone to maintain a personal fork of a project, which contains the original code with some modifications on top, it may be difficult to decide when to advertise this project as a friendly fork intending to take over the place of the original, unmaintained project.

On the one hand, maintaining a fork of an unmaintained project for a long time without doing any advertisement is likely to result in duplicated work, as other persons interested in the project, but who are not aware of the fork, run in the same issues, and prepare their own fixes. Zhou *et al.* [314] studied inefficiencies that may arise mainly because of a lack of awareness of the work that was done in forks.

On the other hand, doing some efforts to advertise the fork (while being clear about the friendly nature of the fork, and the reason for forking) can pay back by bringing an influx of new contributions, from developers that were interested in the project, but were discouraged by the absence of feedback from the maintainer. But it requires effort (going to forums where people usually talk about the project to tell them about the fork), and commitment. The model of community forks that is discussed in the Section 7.4 can help reduce the level of commitment required.

The time to wait until the project is considered unmaintained can also vary depending on community expectations, and is rarely clear to anyone. While SourceForge's "Abandoned Project Takeover" page set a 90-day delay to get an answer (and often effectively processed takeover requests much faster, cf. Footnote 10), CPAN's FAQ [131] informally sets a delay of one year without response before considering transferring maintenance of a module.

How to fork on GitHub? On GitHub, forking a project is as easy as clicking on a button. But, when preparing a hard fork, rather than a development fork, the new maintainer may wonder whether this is the right choice.

By default, forks on GitHub are not meant to take over a project: issues are disabled (but they can easily be switched on), and a prominent link to the original project is displayed under the project's name. Furthermore, code is not searchable in a fork unless the fork has more stars than the original [113] (which can take time to get, given that people rarely remove stars they have given in the past, even when a project is unmaintained).

GitHub does not make discovering maintained forks very easy: the only way to learn about them is to display the fork tree, which is often very large, and to identify the forks that receive pull requests by the fact that they have many forks themselves. When the forks are too numerous, GitHub will not display the full list of forks, and the most important ones may be missing.

For instance, Dagger 2 is maintained at `https://github.com/google/dagger`, a GitHub fork of `https://github.com/square/dagger` where the first version was developed. The new repository is 2,004 commits ahead, and has about twice as many stars (14,199 versus 7,109).

But when displaying the tree of forks, GitHub displays a warning that it is not displaying the full list, and indeed Google's fork is missing. If Dagger 1's own README did not advertise the location of the Dagger 2 sources, someone who was given the link of the original repository will not necessarily learn about the existence of the fork for a while.

The "Lovely forks" browser extension [284] helps developers discover notable forks by querying for them, and showing them below the project's name, where GitHub would display the original project of a fork.

An alternative solution is to create a new repository manually, and to push the content of the original repository in it. It is also possible to contact GitHub staff to remove the fork status from an existing repository. They can also change the base directory in a fork network, but this requires consent from the original owner.[11]

Migrating issues and pull requests. GitHub does not provide any support for easily duplicating the issues from the previous bug tracker. Doing so is nonetheless possible using a tool similar to the one used in Section 4.3.2. Reusing the exact same numbers for existing issues can be done by opening them in the right order, and inserting dummy issues (that can be deleted afterwards) to fill the holes created by pull requests. The advantage of doing this is that the code and the commit message frequently reference issues by their number, and importing preexisting issues ensures that these numbers continue to point to the correct issue. On the other hand, new maintainers might appreciate the ability to manually duplicate only the most important issues that they intend to solve.

If the fork was created using GitHub's fork button, it is also possible to manually recreate pull requests for every pull request that is still opened on the original repository.

How to publish updates to the package? Different registries and different ecosystems have different views regarding the transfer of a package to a new maintainer. Most support voluntary transfers, and some also support transfer to a new maintainer when the previous maintainer is completely unresponsive.

In some package registries, all packages are scoped (cf. "Package naming" in Section 6.4.1), so unless an author explicitly gives access to a new maintainer, there is no way to continue using the package full name, and everyone will have to update their dependencies. On the other hand, this means that the original package will not have a special status compared to the fork in the package index.

In some registries that support both non-scoped and scoped package names, keeping the same base name while adding a scope can be a way of marking the affiliation of the package to

[11]I was told in a private mail by a member of GitHub staff that "we can change the parent of a fork network on our end, but it does require approval from the original owner".

the original one.[12] This is also the technique that is used by the DefinitelyTyped repository (cf. Footnote 4) to publish type definitions corresponding to untyped Javascript packages.

In registries which are based on a shared repository of manifest files, it is technically easy to change the source of a package when publishing an update. The question of whether it gets accepted will depend on the registry's policy. MELPA's policy [190] for instance says that forks will not be accepted except in "extreme circumstances".

Early archives where sources (and sometimes even bug trackers) are located on the platform make it technically even easier to change the maintainer of a package: CTAN, CPAN, CRAN, and PEAR all have documentation regarding unmaintained / orphaned packages. CTAN specifies that modifications to a package should come from the package author or maintainer, new maintainers can be accredited by the current maintainer, but leaves the door open to discussing a solution when a package is unmaintained and the author is unresponsive [65]. CPAN administrators can transfer maintenance of a package to a new volunteer after sufficiently many steps have been taken to reach the previous maintainer and advertise the intent to take over the package [131]. CRAN has a formal orphaning process, after which new volunteers can request to become the new maintainers [63]. Before a package is orphaned, transfer requires written agreement of the previous maintainer. PEAR packages can be marked as unmaintained, and may then be transferred to a new lead maintainer [223].

In general, there is a trust issue associated to allowing a change of maintainer for a package (as this means that someone can update their copy to a new version without realizing the transfer of ownership), but this seems to be part of a much more general trust question in code reuse and package ecosystems. Because of this trust issue, npm is even considering restricting how voluntary transfers can be made [20].

We can see that forking, while often the best solution for the user community, puts a very large cost on the new maintainer, when they do things right and try to organize the community around the new fork. And when forks are created but not properly advertised, it can only lead to duplication of effort. In the next section, I present a solution that alleviates the cost of forking.

7.4 Community forks and community organizations

7.4.1 Community forks for increased sustainability

While hard forks are a possible solution to the problem of unmaintained packages, we have seen that this solution puts a significant cost on the new maintainer. It can also put a significant cost on the user community when the package registry does not make it easy to switch the maintainer of a package, because a possibly large number of users will need to learn about the hard fork, evaluate if it is likely to be viable, and update their dependencies.

[12]As recommended in this Open Source Stack Exchange answer about npm: `https://opensource.stackexchange.com/a/7025/5858`

The transition period, which starts when people become aware that the previous maintainer has disappeared, and which only ends when the user community has massively adopted a hard fork as the new canonical source for the package, is a time during which it is likely that efforts are duplicated, potential contributions as pull requests or bug reports are wasted because they are being ignored or users stop submitting them, the package user base stops growing, or even shrinks, as people look for alternatives to migrate to, or even start their own from scratch, etc.

This cost can be deemed too high, especially if the hard fork itself also has a single maintainer, and thus is likely to suffer from the same issues a few years later. It is natural that the community of users can anticipate this, and will be reluctant to move massively to a hard fork that does not take steps to prevent this scenario to repeat.

A preventive measure would typically be the creation of a community fork. When a package raises sufficient interest, and enough people are motivated to keep maintaining it together, they can host the new sources in a dedicated GitHub organization instead of a personal account, and ensure that at any time, there will be several administrators of this organization, and several persons with credentials to publish a new version of the package.

However, most popular single-maintainer packages are likely to be too small for such a community fork to happen. To facilitate the creation of community forks, a possible model is to host them in a community organization dedicated to the long-term maintenance of important packages. This is the model that I present now.

7.4.2 Community organizations for the long-term maintenance of packages

In this model, a single informal organization is created (typically as an organizational account on GitHub, possibly with some chat rooms / forums / mailing lists, but *a priori* without any legal standing) to host community forks, in a specific ecosystem (typically around a library, a framework, or some specific objective). A place is dedicated to discussing organizational aspects of the community, and to proposing new packages for inclusion (for instance the issue tracker of a meta-repository). The criteria for accepting a package may vary, but generally include having at least one person who is volunteer to maintain the package.

The maintenance may actually be a community effort, but the advantage of having a designated maintainer for a package is to avoid diluting responsibility. However, volunteering to be the principal maintainer of a package is not a long-term commitment. The advantage of hosting the package within a community organization is that the maintenance responsibility can easily be transferred if a maintainer wants to step down or becomes unresponsive, as long as there are responsive organization administrators, and a new volunteer maintainer.

Besides, having a community organization can make collaboration easier, and encourage maintainers to share best practices and help one another. If all the maintainers are given commit access to all projects, a maintainer can easily help maintain another package while its own maintainer is temporarily unavailable.

Hard forking an unmaintained project within such a community organization is likely to be a factor that will help the community accept the hard fork faster, as it creates *de facto* a new central point, and it guarantees against the risks associated with a new single-maintainer package.

Finally, the existence of such a community organization provides an exit strategy for authors of popular packages that would like to step down from maintaining them. They can submit their package for inclusion, and transfer them in the community organization if accepted. Again, inclusion criteria may vary depending on the specific community organization.

7.4.3 The case of elm-community

One of the early instances of this model[13] is the elm-community organization `https://github.com/elm-community/Manifesto`, which was founded in November 2015. I interviewed Ryan Rempel, the founder of elm-community, on July 5th, 2019.

The Elm package ecosystem had already got a culture of package forking and updating when a new version of the Elm language was published but the original package author did not react. This was made easier by the fact that all package names are scoped in the Elm package registry. Consequently, the original package does not have a special status compared to its forks.

However, some non-pure Elm packages (that contained what people used to call native code, and now call kernel code [67], i.e. JavaScript) had to go through a formal "blessing" (whitelisting) process to be published in the Elm package registry. For such packages, forking and updating could not be done so casually, because it would additionally require going through this formal approval process. This specific issue was discussed on the Elm users' mailing list[14] after the author of a widely used package, containing native code, had been unresponsive for two months. During that time, a pull request with a trivial patch required to upgrade the package to the new version of the language (Elm 0.16) had been left unanswered.

Max Goldstein, an active community member, who is now part of Elm's core team, suggested the creation of an "elm-community" GitHub organization "to steward the most important non-official packages". Ryan Rempel, another active community member, who was the author of the unmerged pull request, jumped on the idea, created the organization, forked the package, and submitted a whitelisting request for it, on the same day. Two days later, he created a meta-repository named "Manifesto" in which he described the purpose of the organization in a README, and whose issue tracker served to host organizational discussions, and package adoption requests.

Ryan told me that the reason he had reacted so quickly after the idea was first proposed was because he had viewed this as an opportunity to foster a new form of collaboration, that would be less disciplined, and less centrally controlled than what was common in the Elm community.

[13]Earlier instances include Sous Chefs `https://sous-chefs.org/`, founded in May, 2015, and Vox Pupuli `https://voxpupuli.org`, founded in September 2014.

[14]See: `https://groups.google.com/forum/#!topic/elm-discuss/-GQJkWGdMvg/discussion`.

Indeed, he told me, the Elm community is unusually disciplined for an open source community, around a core team that has very specific ideas about what kind of participation is welcome. All his previous attempts to advocate a more open community had failed, and left him tired. So in this case, he started the community organization without really leaving any time to anyone to discuss the idea, invited five or six people from the beginning, and it turned out to be successful pretty quickly.

Not so long after, the organization gained some repositories that made more sense to develop collaboratively, such as a series of standard library extensions.[15]

The creation of a Manifesto repository was initially meant as a way of explaining the philosophy of the project, but also to allow issues to be used for organizational questions. The word “Manifesto” was slightly provocative and political, but the text of the README avoided any provocative content. According to Ryan, the text was rather abstract at the beginning, and others helped make it more concrete over time. The governance, in particular, remained voluntarily informal.

The idea to have a principal maintainer for each repository, to avoid dilution of responsibility (which leads to issues and pull requests being left unanswered), was introduced about six months later by a member of the organization.[16] This change made explicit the rule that issues and pull requests are normally handled by the repository’s principal maintainer, but in case of unresponsiveness, any other member can step in, and in case of long-term unresponsiveness, the maintainer is changed.

7.4.4 An emerging model

Elm-community is not the only instance of this model, and there are even other instances that predate it.

The Vox Pupuli organization was founded in September 2014 to maintain Puppet modules (initially under the name of puppet-community), and is currently home to over 176 repositories and 139 collaborators. They have a precise migration documentation [293], in which they clearly state that they prefer repository transfer, but can proceed to hard forks when a package author is completely unresponsive. They have also made efforts in recent years to promote their model so that other communities can inspire from it, and create their own [100, 135].

The Sous Chefs organization was founded in May 2015[17] to maintain Chefs “cookbooks” (initially under the name of the Chef Brigade), and had a meta-repository from the start.[18] They also have a clearly documented forking and transfer policy [298, 299]. In particular, their forking

[15]See: `https://groups.google.com/forum/#!topic/elm-discuss/wJPvZUql6v0/discussion`

[16]See the issue `https://github.com/elm-community/Manifesto/issues/16` and the pull request `https://github.com/elm-community/Manifesto/pull/17`.

[17]See the Chef mailing list announcement: `http://lists.opscode.com/sympa/arc/chef/2015-05/msg00091.html`.

[18]The first issue (`https://github.com/sous-chefs/meta/issues/1`) and the first commit (`https://github.com/sous-chefs/meta/commit/efb426f`) were created three days after the initial mailing list announcement.

policy states that hard forks are republished to the package registry under the original name with a "sc-" prefix.

Both organizations are pretty open to any repository transfer from members of the organization.

Dlang-community (`https://github.com/dlang-community/discussions`, founded in December 2016) has stricter inclusion criteria than Vox Pupuli or Sous Chefs: packages are not transferred or created in the organization without it being discussed with other members, and the package being important to the community.

The following organizations have been directly, or indirectly influenced by Elm-community:

- ReasonML-community (`https://github.com/reasonml-community/meta`) was founded in January 2017 under the name Buckletypes (following the model of the TypeScript DefinitelyTyped organization, cf. Footnote 4). It was renamed in July 2017, and got a meta-repository influenced by Elm-community in January 2018.[19] However it has failed to get clear adoption guidelines, and has recently failed to adopt the very popular graphql_ppx package, after more than six months of unresponsiveness from its author, and many such suggestions in the ReasonML Discord chat.

- Coq-community (`https://github.com/coq-community/manifesto`, launched in July 2018 although I created the organization in December 2017) was directly influenced by Elm-community. I present it in Section 7.5.2.

- OCaml-community (`https://github.com/ocaml-community/meta`, founded in August 2018) was influenced by Coq-community and Elm-community. Similarly to Dlang-community, it only accepts popular OCaml packages.

- React-native-community (`https://github.com/react-native-community/.github`) was founded in July 2016, but got a "renaissance" period starting in December 2018 that was influenced by OCaml-community.[20]

Some organizations are similarly structured, and intend to facilitate the maintenance of projects after their maintainer has left or lost interest, but do not explicitly support forking already unmaintained projects. Examples include the F# Community Incubation space (`https://github.com/fsprojects/FsProjectsAdmin`, founded in November 2013), Electron Userland (`https://github.com/electron-userland/welcome`, founded in February 2016), the Elytra group (`https://github.com/elytra`, founded in May 2016), which does not have a meta-repository or general guidelines, but does list the maintainers of each repository in their description, and advertise when a module is looking for a new maintainer. The Fluent Plugins Nursery (`https://github.com/fluent-plugins-nursery/contact`, founded in September

[19]See `https://github.com/glennsl/reasonml-community-meta-proposal` and `https://github.com/reasonml-community/meta/issues/1`.

[20]See: `https://github.com/react-native-community/discussions-and-proposals/issues/63`.

2016) is explicitly intended for plugins that are not maintained by their original author, but also states “we don’t want to fork original authors’ ”.

In the following, I present the process I used to find about the organizations that I listed in this section (cf. the corresponding Jupyter notebook [319]).

First, I listed 75 keywords that could be expected to appear in the name or the description of such organizations. This included keywords expressing collaborative work such as “collaboration”, “collective”, “community”, “contribution”, “give”, “help”, “maintain”, “participate”, “support”, but also keywords expressing what is being worked on such as “app”, “component”, “ecosystem”, “extension”, “library”, “module”, “package”, “plugin”, “repository”.

I used GitHub’s search, via the GraphQL API (cf. Section 2.2.1), to query for organizations which matched one of these keywords, and which had at least 5 repositories. GitHub’s search only gives access to 1000 results, so when the number of results was above this limit, I further split the search using language filters (I searched for organizations containing repositories written in a number of popular languages such as Javascript, Java, Python, etc., and then all the organizations that did not contain repositories written in these languages).

For each organization, I queried for the login, name, description, website URL, creation date, number of public members, and, for its most starred repository, the number of stars, and the number of assignable users.

This first step yielded over 30,000 organizations (this is close to 15% of all GitHub organizations with at least 5 repositories). Thanks to the use of the GraphQL API, the number of queries was rather limited, since I could fetch the information for as many as 50 organizations in a single request.

The second step consisted in applying some filters on the results. Since I was only interested in community organizations, I filtered out the ones that had less than 10 public members, or 10 assignable users on the most starred repository (as a way to estimate the number of collaborators for organization with mostly private members). Since I was interested in organizations maintaining important packages, I also filtered out the ones whose most popular repository had less than 10 stars.

The third step was intended to further reduce the list to organizations that have received repository transfers. Unfortunately, GitHub does not give access to this information. I used a trick to find these organizations nonetheless, which consisted in comparing the creation date of the organization to the creation date of its repositories. If an organization contains repositories that predate the creation of the organization, they have necessarily been transferred. Obviously, the converse is not true, so it could have resulted in underestimating the number of transferred repositories.

For each organization, I searched for all its repositories that were created before the organization’s creation date, and recorded the number of results.

Finally, I manually browsed the list of 938 organizations with at least two transferred repositories predating its creation. I used the name and the description of the organization to infer its purpose. I also eliminated many organizations whose description was something like "community packages for X" when the URL of the organization was actually the website of X. Indeed, this type of organization is too frequent, and the reuse of the main product's website is a good indicator of the absence of a website, or meta-repository, specific to the community organization.

When an organization seemed to correspond to the type I was searching, I opened its website, or GitHub page, and looked for more information about it. It was frequent to find organizations with many repositories but no public description of its principles, and whether the organization accepted new members or new projects.

I am well aware that applying this series of filters is very likely to have resulted in missing out organizations that still fit the model I presented in Section 7.4.2. Nevertheless, the number of examples that I found, and the absence of relationship between many of them (in particular, between Elm-community and the two organizations that were founded before) leads me to think that this model of organization is naturally emerging in open source package ecosystems, in answer to the recurring problem of single-maintainer libraries that I presented in Section 7.2. The fact that such organizations frequently got inspiration from one another shows that, while the need for such an organization is natural, the exact way of structuring it is not. Therefore, this contribution is important because, by surveying existing instances of this model, this can bring useful information to practitioners wondering about the opportunity of founding such organizations, and how to structure them.

7.4.5 How does it compare to earlier models?

The idea of sharing a common infrastructure to develop several projects together is not new.

For instance, PEAR [222] was PHP's first package manager and registry, but it was also much more. PHP packages had to go through a formal submission process to get accepted. Once they were accepted, they got an SVN repository, and a bug tracker. Finally, when a package was left unmaintained by its author, a new maintainer could be appointed [223].

However, there was an important difference in the motivations of authors applying to get their packages accepted in PEAR, compared to authors, or interested users, proposing a package to a community organization today. As PEAR was PHP's only package manager and registry at the time (until support for alternative registries was introduced, starting in 2005[21]), authors submitted their packages to get it distributed, rather than for the shared maintenance model. For some maintainers, the model and infrastructure were convenient, but others preferred to host their projects elsewhere.

Today, the PEAR model is partially reproduced whenever a GitHub organization is created mostly to serve as a package index, and when package authors only transfer their packages to

[21]Cf. `http://php-pear.1086190.n5.nabble.com/Questions-about-PEAR-amp-Composer-td36854.html`.

get more visibility.

Another well-known example is the Apache Foundation, which hosts many open source projects, and provides shared processes and infrastructure. The main difference with the model of community organization that I presented is that the Apache Foundation only accepts larger projects (for instance, all the projects have several maintainers, and their own mailing lists [13]). The Apache Foundation model is therefore not suited to solve the issue of single-maintainer packages, and is not ready to fork unmaintained packages (despite the original Apache web server being a community fork of an unmaintained project).

7.4.6 Open issues, future work

I have presented a model of community organization for the collaborative, long-term maintenance of packages belonging to a given ecosystem, and I have identified numerous instances of this model. However, the method I used to find them cannot be considered to provide an exhaustive list.

The use of GitHub Search to list possible candidates is good for exploratory work, but cannot be used beyond that: I have observed that the results obtained are very unstable when trying to fetch them again, and I have also found a number of bugs in GitHub's search filters. According to GitHub staff, this is because the search index is sometimes out of date.

Finding more examples will require coming up with more precise criteria to detect automatically this kind of organizations, probably starting from the complete list of 2,000,000 GitHub organizations. Then, we could try to identify which characteristics of an ecosystem's structure favor the emergence of such organizations.

Given that this is an emerging model, each instance is unique, and it would be useful to come up with a number of parameters that can be used to describe them. Interviewing founders and participants would be helpful in that regard. Then, the next step would be to identify which of these parameters are associated with successful community organizations. For instance, it would seem that documenting the adoption process is helpful to ensure that people know what to do when a useful package in the ecosystem has been abandoned and would be a candidate for adoption, whereas the absence of such guidelines can lead to a lack of reactivity, and duplication of effort with multiple people starting forks (as happened with the graphql_ppx library in the ReasonML ecosystem).

Finally, it seems important to systematically assess the impact of such community organizations on an ecosystem, and in particular on the packages that get transferred or forked in the organization. For instance, Zhou *et al.* [314] studied inefficiencies in fork-based development, such as duplication of work or community fragmentation. We could try to evaluate whether the presence of a community organization can help reduce these inefficiencies, and under which conditions. This would provide concrete incentives to practitioners to create more instances of this model.

7.5 Package maintenance in the Coq ecosystem

7.5.1 Historical context

As I explained in Section 6.5.1, the distribution channel for Coq packages used to be based on a submission process to join a set of “contribs”. With the introduction of an opam-based package registry, this submission process was shut down. Another use of the contribs had been for compatibility testing, and this was also replaced by a new model, presented in Section 3.6.2. However, there remained a use of the contribs that had not been replaced, which was that once a Coq package was in the contribs, it was kept compatible with new versions of Coq by the Coq development team.

This was actually very useful because many Coq packages would not have been continuously maintained otherwise. Indeed, many Coq packages are developed by PhD students who are likely to move on to other things after they graduate.

Furthermore, many Coq packages are paper artifacts that showcase a new technique, or formalize a mathematical theorem, and are not really meant to be reusable components, but this does not make them less interesting, or less worth maintaining. Being able to execute step-by-step a Coq proof plays a large role in being able to understand it, and researchers that want to understand in depth a paper several years after it was published may have difficulties doing so, if the artifact was not maintained for compatibility with new versions of Coq during this time.

Besides, even though the submission process had been shut down, there still were more than 150 contribs predating the introduction of the opam-based registry. These contribs had been moved out of the historic SVN repository in which they were maintained, into a number of git repositories in 2015,[22] so that they could be published in the Coq package registry more easily [53]. These git repositories were moved to a special GitHub organization in 2016.[23] However, since the contribs had stopped being used for compatibility testing, most Coq developers had lost interest in maintaining them.

7.5.2 The creation of coq-community

I proposed to create an organization dedicated to the long-term maintenance of Coq packages, and based on the Elm-community model to solve these issues. In particular, the goal was to involve the user community in the maintenance of packages, including former contribs.

I first presented the idea to Pierre Castéran and Hugo Herbelin in the end of 2017, then to the Coq development team in the beginning of 2018. I used the following months to prepare the content of the Manifesto repository with the intent to announce the project at the Coq workshop that was held in Oxford in July 2018.

[22]Cf. `https://github.com/coq-contribs/coq-contribs/commit/f5814d5`.
[23]Cf. `https://github.com/coq-contribs/coq-contribs/commit/976667c`.

Figure 7.1: The logo of Coq-community. This logo was created by Aras Atasaygin as part of the Open Logos project (`http://openlogos.org/`).

The manifesto spelled out three objectives: the long-term maintenance of interesting Coq packages; the creation of collaborative documentation; and the creation of an editorial board to put forward the most valuable packages.

The first objective has been quite successful since little more than a year after its creation, the Coq-community organization now hosts 20 projects (among them, 11 were transferred from the coq-contribs organization) maintained by 15 principal maintainers (myself not included). The organization has three owners, which are among the most active members.

Contribs are only transferred to the Coq-community organization when they find a volunteer maintainer. A successful way of recruiting such volunteers has been to propose to the people submitting pull requests on contrib repositories whether they wanted to become the contrib's maintainer within Coq-community.

The second and the third objective have not yet taken shape.

The collaborative documentation objective is based on an idea of Pierre Castéran to create tutorials based on concrete Coq projects, to demonstrate advanced formalization techniques. While the initial writing of such a tutorial is a task that can only been undertaken by a single individual, or a small group, the interest of having it live in an organization like Coq-community will be to help maintain and improve the tutorial over time. Furthermore, once a first example is available, we intend to propose to other experienced Coq users to contribute their own.

The objective of creating an editorial board was meant as an answer to a recurring critic that the contribs used to receive, which was that contribs were accepted without any curation process regarding their quality. However, such critic has not been expressed recently, so the need to create this editorial board has not been pressing.

A fourth, implicit, objective was to be a hub for exploration and documentation of maintenance best practices. This has been successful with the development of a set of templates that can be used to generate a Travis CI configuration file, an opam file, a Nix file, and a README, based on a meta file stored in YAML format [25]. The creation of Coq-community has also triggered and contributed to efforts by Erik Martin-Dorel and myself to come up with efficient methods of setting up CI for Coq projects (as I mentioned in Section 6.5.2).

7.5.3 Open issues, future work

A lot remains to do to scale Coq-community, besides making progress on the second and third objectives. We need to provide maintainers better maintenance guidelines, and better automation. This is not only useful to maintainers within Coq-community, but also in the rest of the Coq ecosystem, as this is an opportunity to identify pain points in the maintenance of Coq packages, and to solve them with documented best practices, and tools allowing more automation. Some ideas could be found by observing how larger and older community organizations, such as Vox Pupuli, proceed.

7.6 Conclusion

Modern and attractive package ecosystems are made of a multitude of small and large open source libraries. When a popular package is depended upon by many projects, but has only a single maintainer, it creates a risk, of the maintainer suddenly becoming unresponsive, and the package not being updated anymore. I have shown that a large proportion of popular packages are single-maintainer packages, and the mitigation models that users of these packages can adopt in such situations of unresponsiveness. Among them, the friendly fork model is often the best for the community, but it can be costly to start a fork, and it does not help very much if the new fork is also a single-maintainer package. This is why users can create community organizations for the collaborative, long-term maintenance of an ecosystem's packages. Such organizations can reduce the cost of forking, while creating better guarantees for the future of the fork. I have shown that this model of community organizations has emerged in a number of package ecosystems. Finally, I have presented the Coq-community organization that I founded on this model for the Coq ecosystem.

CHAPTER 8

CONCLUSION AND OPEN ISSUES

8.1 Conclusion

At this point, I hope I have convinced the reader of the importance of addressing software engineering challenges, during the development, maintenance and evolution of long-lived academic software systems, especially programming languages. At least, it would seem that I have managed to convince the members of the development team of Coq —a programming language to prove theorems and properties of programs, on which I have focused during the three years of my PhD, and around which I gravitated during the two previous years.

Coq is the result of a long-line of research. The development of this proof system started in the 1980s at Inria, but its origins can even be traced further back to the invention of interactive proof assistants in the 1970s, to seminal work on automated theorem proving in the 1960s, to the invention of functional programming in the 1950s, and even to the first attempts to formalize mathematical logic in the beginning of the 20th century [59].

Coq, and other proof assistants created around the same time, have made it possible to actually verify that the proofs of very complex mathematical theorems were correct, and that certain critical or tedious programs did behave as they were supposed to. Challenges that users of such proof assistants have encountered when working on large-scale projects gave rise to a new domain of research called proof engineering [244]. But these successes also popularized the use of proof assistants, resulting in an increased number of users and in bigger projects, whose authors expressed new needs such as stability and compatibility. The needs of new users, and of authors of big projects, but also the age of the Coq system, and the will to open the development to contributors all around the world, have all led to much more classical software engineering issues, related to software development processes, software maintenance and evolution, open source development, and software ecosystems.

I personally started to get interested in Coq because I had ambitions to explore massively collaborative theorem proving [315] (as part of a more general taste for open collaboration around science [317]). But after I first met my PhD advisor, I shifted my focus on making Coq easier to use for mathematicians, because this would be a prerequisite to allow large numbers of people to use Coq collaboratively [316]. And after I started contributing to the code of Coq, I began identifying many software engineering issues that needed to be solved to allow Coq to evolve more efficiently, and to better address the needs of its current and future users. I was not the only one in the development team to put efforts on these questions: I have already cited the numerous contributions of, among others, Maxime Dénès, Emilio Gallego Arias, and Gaëtan Gilbert. But I decided to go further by turning this into academic research. Incidentally, I went back that way to the study of open collaboration.

I adopted the approach of insider action research, which I have presented in Section 1.4. It means that I have made concrete contributions to Coq's development tools and processes, and to the organization of its ecosystem, by collaborating with other developers and users. But also that I have contributed to the scientific literature, by reporting on this experience, and conducting rigorous empirical evaluation to validate some actions.

My contributions, listed in more details in Section 1.5 range from identifying new research questions, to proposing generic solutions that can address some of them, and conducting empirical analyses based on mined software repository data. In a first part, composed of Chapters 3, 4, and 5, I have presented a historical overview and addressed concerns related to the maintenance and evolution of the Coq system: the switch from push-based to pull-based development, which facilitated an increased involvement of external contributors, and an increased focus on the quality of integrated changes; the switch of the bug tracker from Bugzilla to GitHub, which led to more bug tracking activity, and more external participants in the discussions; the switch to shorter release cycles, which was associated with challenges of efficiency and efficacy of release management. In a second part, composed of Chapters 6 and 7, I have focused on concerns at the level of package ecosystems, regarding both package distribution and maintenance, drawing inspiration from other package ecosystems to improve the one of Coq.

8.2 Open issues and future work

I identified and already listed many open issues, that I would like to address in the future, related to the topics of the previous chapters (cf. Sections 2.5.4, 3.6.6, 3.7.1, 5.4.3, 5.5.4, 5.6, 6.5.2, 7.4.6, and 7.5.3). In the remainder of this conclusion, I present some additional open issues which are out of the scope of these chapters.

8.2.1 Documentation

Documentation is a critical asset to make a software system accessible (especially a programming language, and all the more a complex proof language such as Coq). Fortunately, Coq is better and better documented, with the user community having contributed numerous tutorials and books. From the perspective of core developers, the main documentation objectives are to continually improve the reference manual [59], which must be clear and exhaustive, because experienced users frequently refer to it (especially as most advanced features and corner cases are much less frequently documented in tutorials). Beyond clarity and exhaustiveness, a challenge is consistency with the current state of the system. Tremendous progress has already been made with a culture shift toward documenting new features or updating the documentation when conducting user-visible changes *before* integrating such changes. This culture shift began with the adoption of pull-based development, and was further emphasized thanks to the introduction of the pull request template (cf. Section 3.5.2).

The reference manual has received major improvements recently, starting with the adoption of a new format based on Sphinx [39].[1] Many things remain to do, including designing a better structure so that all parts of the system are properly covered, but also improving the build infrastructure of the manual to add more static guarantees on the consistency of the documentation with respect to the code. For instance, work is in progress to automatically verify that the documented grammar of the language corresponds to the one that is actually defined in the code.

8.2.2 Bug triaging

While we have shown in Chapter 4 that the switch of bug tracker had a positive effect on bug tracking activity, leading in particular main developers to report more systematically the bugs they encounter, we have not measured the effect on the bug fixing rate, the bug resolving rate, or more generally on bug triaging activities. Even assuming that the impact on the bug resolving rate was positive, this resolving rate is still too low, with the number of open issues steadily increasing (it was just above 1,000 at the date of the bug tracker switch two years ago, and is now just above 2,000, even if a quarter of the migrated open issues have been resolved in the meantime). This high number does not necessarily mean anything wrong, as issues are now more systematically opened as long-term reminders of possible improvements or unresolved questions. But it also makes it harder to browse through open issues. To compensate, it would be very useful to have a good triaging model, which would allow developers to know the status of an issue at a glance, and quickly find those that are relevant to them. A lot of previous work exists on automatically triaging issues, most particularly on automatically assigning issues to relevant developers or teams (e.g. the recent work of Sarkar *et al.* [254]). However, most of the techniques and tools that were developed are intended for a commercial development context (and those

[1]This followed a proposal by Clément Pit-Claudel in 2016 to port the manual to this format, using a custom Sphinx extension that he had created to support the specific needs of the Coq manual [228].

intended for open source are only good for basic triaging tasks [158]): in open source, where most work is voluntary, assigning issues to developers without their consent does not work, and just leads to assignment losing meaning. Involving the community in the bug triaging process, to cope with many incoming, and backlog issues, is a challenge that we have to address in order to maintain a satisfying bug tracker state, without falling for the easy solution of using a bot to close stale issues.[2]

8.2.3 Separating design discussions from implementation discussions

Many developer communities have observed that mixing discussion on design and on specific implementation details is not generally a good idea. That is why some open source projects have a process for reviewing proposed design documents, separately and before reviewing the corresponding implementation. Some major programming languages such as Python and Rust have adopted such process to review all major feature proposals, as well as proposals to change development processes. The Python model is called "Python Enhancement Proposals" (PEP) [287, 297], and the Rust model [251] reuses the "Request For Comments" (RFC) terminology from IETF. Both processes use a GitHub repository for contributors to propose new design documents. The Rust RFC reviewing process takes place via pull requests, and when consensus seems to be found, the RFC enters a final comment period (FCP) to validate a proposed decision. If the RFC is officially approved, the corresponding pull request is merged. On the other hand, given that the PEP process is much older than GitHub and the invention of pull requests, a PEP can be committed directly to the repository, or merged via a pull request, before it is officially approved. The review and decision process takes place afterward.

Coq got "Coq Enhancement Proposals" (CEP) in 2016 under the impulse of Enrico Tassi. It was further refined in 2017 by taking inspiration from Rust and associating the integration of a pull request to the approval of the CEP. However, it has not really taken off, in particular because most proposed CEPs have eventually staled instead of being accepted or rejected, and also because major design decisions have continued to be proposed and approved through code pull requests. The NixOS organization, which has also an RFC repository, suffered from the same issues, but has recently managed to address them by adopting a complex workflow with shepherd teams and FCPs (inspired by the Rust model) [120]. The questions for the Coq project are: How to adopt a similar successful workflow (probably based on FCP and shepherds) but at a smaller team scale? And how to motivate (both core and external) contributors to go through this process rather than submitting feature pull requests directly? Can we identify in which cases this extra step is worthwhile?

[2]More than a thousand GitHub users and organizations are using Probot Stale [162] to automatically close stale issues and pull requests.

8.2.4 Governance

As of today, the governance model of Coq has become unclear. This is rarely a problem because, as in most open source projects, the default decision process is based on consensus (with silence being implicit consent) [95, Chapter 4]. Pull requests are not merged when the need for more discussion is expressed, and such discussion can happen during regularly held working groups. However, when no consensus is still to be found, what is our decision model? We have a designated project leader, and a designated core team. Do we follow a benevolent dictator model with the project leader taking decisions ultimately, or do we follow a voting based model? At least we have made progress on defining the release management process, with a release manager who acts as the release owner, i.e. the person who gets to decide what goes in a release (cf. Chapter 5). We have also introduced a code of conduct,[3] and have a designated enforcement team. However, we are still missing a clear process for choosing the next release manager, adding and removing core team members, adding and removing code owners (cf. Section 3.7), adding and removing members to the code of conduct enforcement team, and changing of project leader.

[3]Code of conducts are an emerging phenomenon in open source projects, intended to provide safe and inclusive communities, and to deal with conflicts and negative feelings [280].

Bibliography

[1] G. Abaee and D. Guru, *Enhancement of bug tracking tools; the debugger*, in Proceedings of the 2nd International Conference on Software Technology and Engineering, vol. 1, IEEE, 2010, pp. V1–165.
4.2.1

[2] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, *Why do developers use trivial packages? An empirical case study on npm*, in Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 385–395.
7.2

[3] A. Abdellatif, K. Badran, and E. Shihab, *A repository of research articles on software bots*.
http://papers.botse.org.
2.5.3

[4] Z. Abou Khalil, *Studying the impact of policy changes on bug handling performance*, in Proceedings of the 35th International Conference on Software Maintenance and Evolution, IEEE, 2019.
4.1

[5] B. Aebischer and L. M. Hilty, *The energy demand of ICT: A historical perspective and current methodological challenges*, in ICT Innovations for Sustainability, 2015.
3.6.6.2

[6] O. Alders, R. Stauner, et al., *MetaCPAN API docs: v1*.
https://github.com/metacpan/metacpan-api/blob/90e63f7/docs/API-docs.md.
Last accessed September 17th, 2019.
6.4.1

[7] A. Allen, C. Aragon, C. Becker, J. Carver, A. Chis, B. Combemale, M. Croucher, K. Crowston, D. Garijo, A. Gehani, et al., *Engineering academic software*, in Dagstuhl Manifestos, vol. 6, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
1.3, 2.4

[8] H. ALRUBAYE, M. WIEM MKAOUER, AND A. OUNI, *MigrationMiner: An automated detection tool of third-party Java library migration at the method level*, in Proceedings of the 35th International Conference on Software Maintenance and Evolution, IEEE, 2019.
7.3.1

[9] ANACONDA, INC., *Anaconda*.
https://www.anaconda.com/, 2012.
Last accessed September 20th, 2019.
6.3.2

[10] J. D. ANGRIST AND J.-S. PISCHKE, *Mostly harmless econometrics: An empiricist's companion*, Princeton university press, 2008.
2.3.2.1

[11] ——, *Mastering 'metrics: The path from cause to effect*, Princeton University Press, 2014.
2.3.2.1

[12] J. ANVIK, L. HIEW, AND G. C. MURPHY, *Coping with an open bug repository*, in Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange, ACM, 2005, pp. 35–39.
4.1

[13] APACHE SOFTWARE FOUNDATION, *How it works*.
https://www.apache.org/foundation/how-it-works.html.
Last accessed September 21st, 2019.
7.4.5

[14] ——, *Apache Subversion*.
https://svn.apache.org/viewvc/subversion/trunk/, 2000–2019.
Last accessed September 20th, 2019.
1, 3.3

[15] APPVEYOR SYSTEMS INC., *AppVeyor*.
https://www.appveyor.com/.
Last accessed September 9th, 2019.
3.6.1, 3.6.4, 5.1

[16] D. ASPINALL, *Proof General: A generic tool for proof development*, in International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2000, pp. 38–43.
4.4.5.2

[17] J. ATWOOD, R. WARD, AND S. SAFFRON, *Discourse*.
https://www.discourse.org/, 2013–2019.

Last accessed September 9th, 2019.
3.4.1

[18] G. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik, *On the abandonment and survival of open source projects: An empirical investigation*, in Proceedings of the 13th International Symposium on Empirical Software Engineering and Measurement, IEEE/ACM, 2019.
7.2.2.1

[19] A. Baldwin, *Details about the event-stream incident*.
`https://blog.npmjs.org/post/180565383195`, 2018.
Last accessed July 23rd, 2019.
3

[20] ——, *The security risks of changing package owners*.
`https://blog.npmjs.org/post/182828408610`, 2019.
Last accessed September 4th, 2019.
7.3.3.2

[21] M. P. Barnson et al., *The Bugzilla guide*.
`http://ftp.task.gda.pl/mirror/sunsite.unc.edu/pub/Linux/docs/linux-doc-project/bugzilla-guide/Bugzilla-Guide.pdf`, 2001.
Last accessed September 12th, 2019.
4.3.1

[22] B. Barras, H. Herbelin, and O. Desmettre, *Translation from Coq V7 to V8*.
`https://coq.inria.fr/distrib/V8.0pl3/coq-8.0pl3-translator.tar.gz`, 2004.
Last accessed September 10th, 2019.
3.6.2

[23] O. Baysal, R. Holmes, and M. W. Godfrey, *Situational awareness: personalizing issue tracking systems*, in Proceedings of the 35th International Conference on Software Engineering, IEEE/ACM, 2013, pp. 1185–1188.
4.2.1

[24] ——, *No issue left behind: Reducing information overload in issue tracking*, in Proceedings of the 22nd International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 666–677.
4.2.1

[25] O. Ben-Kiki, C. Evans, and I. döt Net, *YAML Ain't Markup Language version 1.2*.
`https://yaml.org/spec/1.2/spec.html`, 2009.

Last accessed August 15th, 2019.
2.5.3, 5.5.4, 6.4.2, 7.5.2

[26] A. BERESTOVSKY, *bugzilla2github*.
`https://github.com/berestovskyy/bugzilla2github/`, 2018.
Last accessed November 29th, 2019.
4.3.2

[27] Y. BERTOT AND P. CASTÉRAN, *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*, Springer Science & Business Media, 2004.
1.3

[28] N. BETTENBURG, R. PREMRAJ, THOMAS ZIMMERMANN, AND S. KIM, *Duplicate bug reports considered harmful... really?*, in Proceedings of the 24th International Conference on Software Maintenance, IEEE, 2008, pp. 337–345.
4.1

[29] T. F. BISSYANDÉ, D. LO, L. JIANG, L. RÉVEILLERE, J. KLEIN, AND Y. LE TRAON, *Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub*, in Proceedings of the 24th International Symposium on Software Reliability Engineering, IEEE, 2013, pp. 188–197.
3.5.1, 4.2.2

[30] B. BLEIKAMP, *Protected branches and required status checks*.
`https://github.blog/2015-09-03-protected-branches-and-required-status-checks/`, 2015.
Last accessed August 23rd, 2019.
3.7

[31] ——, *Issue and pull request templates*.
`https://github.blog/2016-02-17-issue-and-pull-request-templates/`, 2016.
Last accessed August 18th, 2019.
3.5.2, 4

[32] W. BOATS, *Organizational debt*.
`https://boats.gitlab.io/blog/post/rust-2019/`, 2018.
Last accessed September 26th, 2019.
1.3

[33] C. BOETTIGER, *A drat repository for rOpenSci*.
`https://ropensci.org/blog/2015/08/04/a-drat-repository-for-ropensci/`, 2015.

Last accessed August 30th, 2019.
6.4.1

[34] C. BOGART, A. FILIPPOVA, J. HERBSLEB, AND C. KASTNER, *Culture and breaking change: A survey of values and practices in 18 open source software ecosystems*.
`https://doi.org/10.1184/R1/5108716.v1`, 2017.
6.1, 6.4.1

[35] C. BOGART, C. KÄSTNER, J. HERBSLEB, AND F. THUNG, *How to break an API: cost negotiation and community values in three software ecosystems*, in Proceedings of the 24th International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 109–120.
6.1

[36] G. T. BORTIS, *PorchLight: A Tag-based Approach to Bug Triaging*, PhD thesis, University of California at Irvine, 2016.
4.2.1

[37] F. BOUR, T. REFIS, AND G. SCHERER, *Merlin: a language server for OCaml (experience report)*, Proceedings of the ACM on Programming Languages, 2 (2018).
2.5.1

[38] J. BOXER, *Add reactions to pull requests, issues, and comments*.
`https://github.blog/2016-03-10-add-reactions-to-pull-requests-issues-and-comments/`, 2016.
Last accessed October 8th, 2019.
3.4

[39] G. BRANDL, *Sphinx, Python documentation generator*.
`http://www.sphinx-doc.org/`, 2008–2019.
Last accessed August 17th, 2019.
4.4.5.2, 5.5.4, 8.2.1

[40] S. BREU, R. PREMRAJ, J. SILLITO, AND THOMAS ZIMMERMANN, *Information needs in bug reports: improving cooperation between developers and users*, in Proceedings of the 2010 Conference on Computer Supported Cooperative Work, ACM, 2010, pp. 301–310.
4.1

[41] C. BRINDESCU, M. CODOBAN, S. SHMARKATIUK, AND D. DIG, *How do centralized and distributed version control systems impact software changes?*, in Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 322–333.
3.2.2

[42] R. BUYYA, M. PATHAN, AND A. VAKALI, *Content delivery networks*, vol. 9, Springer Science & Business Media, 2008.
6.3.2

[43] L. BYRON, *GraphQL: A data query language*.
https://engineering.fb.com/core-data/graphql-a-data-query-language/, 2015.
Last accessed August 13th, 2019.
2.2.1, 6.4.1

[44] CABAL TEAM, *Haskell package versioning policy*.
https://pvp.haskell.org/.
Last accessed September 1st, 2019.
6.4.2

[45] ——, *Cabal user guide*.
https://www.haskell.org/cabal/users-guide/index.html, 2003–2017.
Last accessed September 1st, 2019.
6.4.4

[46] J. CABOT, J. L. C. IZQUIERDO, V. COSENTINO, AND B. ROLANDI, *Exploring the use of labels to categorize issues in open-source software projects*, in Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 550–554.
3.5.1

[47] CANONICAL LTD., *Snapcraft: The app store for Linux*.
https://snapcraft.io/, 2015–2019.
Last accessed September 18th, 2019.
6.5.2

[48] A. CELIK, K. PALMSKOG, AND M. GLIGORIC, *ICoq: Regression proof selection for large-scale verification projects*, in Proceedings of the 32nd International Conference on Automated Software Engineering, IEEE/ACM, 2017, pp. 171–182.
3.6.6.2

[49] A. CESAR BRANDÃO GOMES DA SILVA, G. DE FIGUEIREDO CARNEIRO, F. BRITO E ABREU, AND M. PESSOA MONTEIRO, *Frequent releases in open source software: A systematic review*, Information, 8 (2017), p. 109.
5.1

[50] T. CHAJED, T. ZIMMERMANN, ET AL., *Guide to contributing to Coq*.
https://github.com/coq/coq/blob/396f814/CONTRIBUTING.md, 2017–2019.

Last accessed September 9th, 2019.
3.4, 3.6.6.3

[51] CIRCLECI, *CircleCI*.
`https://circleci.com/`, 2011–2019.
Last accessed September 9th, 2019.
3.6.1, 3.6.4, 5.1

[52] G. CLARET, *Use opam for Coq*.
`http://coq-blog.clarus.me/use-opam-for-coq.html`, 2014.
Last accessed September 2nd, 2019.
6.5.1

[53] ——, *OPAM for Coq*, HAL preprint hal-02292537, Inria, 2015.
6.5.1, 7.5.1

[54] J. COELHO AND M. T. VALENTE, *Why modern open source projects fail*, in Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 186–196.
3.5.2

[55] D. COGHLAN AND T. BRANNICK, *Doing action research in your own organization*, SAGE Publications, 2019.
1.4

[56] C. COHEN, *Agenda of the April 23rd 2019 meeting 9h30 to 12h30*.
https://github.com/math-comp/math-comp/wiki/Agenda-of-the-April-23rd-2019-meeting-9h30-to-12h30, 2019.
Last accessed August 26th, 2019.
5.5.3

[57] E. CONSTANTINOU, A. DECAN, AND T. MENS, *Breaking the borders: an investigation of cross-ecosystem software packages*, in 17th Belgium-Netherlands Software Evolution Workshop (BENEVOL), Delft, Netherlands, December 2018.
2.2.2

[58] CONVENTIONAL CHANGELOG TEAM, *Conventional changelog*.
`https://github.com/conventional-changelog/conventional-changelog`.
Last accessed August 26th, 2019.
5.1, 5.5.1, 5.5.3

[59] COQ DEVELOPMENT TEAM, *The Coq reference manual*.
`https://github.com/coq/coq/releases/download/V8.10.0/coq-8.10.0-reference-manual.pdf`, 1984–2019.

Last accessed October 10th, 2019.
3.6.2, 8.1, 8.2.1

[60] J. CORBET, *The Ottawa kernel summit, day two*.
`https://lwn.net/Articles/3467/`, 2002.
Last accessed September 15th, 2019.
5.1

[61] J. CORBET, *The case of the supersized shebang*.
`https://lwn.net/Articles/779997/`, 2019.
Last accessed August 25th, 2019.
5.4.1

[62] V. COSENTINO, J. L. C. IZQUIERDO, AND J. CABOT, *A systematic mapping study of software development with GitHub*, IEEE Access, 5 (2017), pp. 7173–7192.
3.1, 3.2.1

[63] CRAN REPOSITORY MAINTAINERS, *CRAN repository policy*.
`https://cran.r-project.org/web/packages/policies.html`.
Last accessed August 30th, 2019.
13, 7.3.3.2

[64] D. CROCKFORD, *The JavaScript object notation (JSON) data interchange format*, Request for Comments, Internet Engineering Task Force, 2017.
2.5.1, 6.4.2

[65] CTAN TEAM, *How can I take authorship of a package?*
`https://ctan.org/help/become-author`.
Last accessed September 4th, 2019.
7.3.3.2

[66] CVS TEAM, *Concurrent versions system*.
`https://www.gnu.org/software/trans-coord/manual/cvs/cvs.html`, 1990–2005.
Last accessed September 20th, 2019.
3.3

[67] E. CZAPLICKI, *"Native code" in 0.19*.
`https://discourse.elm-lang.org/t/native-code-in-0-19/826`, 2018.
Last accessed September 6th, 2019.
7.4.3

[68] D. A. DA COSTA, S. MCINTOSH, C. TREUDE, U. KULESZA, AND A. E. HASSAN, *The impact of rapid release cycles on the integration delay of fixed issues*, Empirical Software Engineering, 23 (2018), pp. 1–70.

5.2, 5.6

[69] L. DABBISH, C. STUART, J. TSAY, AND J. HERBSLEB, *Social coding in GitHub: transparency and collaboration in an open software repository*, in Proceedings of the 15th Conference on Computer Supported Cooperative Work, ACM, 2012, pp. 1277–1286.
3.2.1

[70] K. DAIGLE, *GitHub Actions: built by you, run by us*.
`https://github.blog/2018-10-17-action-demos/`, 2018.
Last accessed August 15th, 2019.
2.5.3

[71] J. L. DAVIDSON, N. MOHAN, AND C. JENSEN, *Coping with duplicate bug reports in free/open source software projects*, in Proceedings of the 2011 Symposium on Visual Languages and Human-Centric Computing, IEEE, 2011, pp. 101–108.
4.1

[72] B. DE ALWIS AND J. SILLITO, *Why are software projects moving from centralized to decentralized version control systems?*, in Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering, IEEE, 2009, pp. 36–39.
3.1, 3.2.2

[73] A. DECAN AND T. MENS, *What do package dependencies tell us about semantic versioning?*, IEEE Transactions on Software Engineering, (2019).
6.1

[74] A. DECAN, T. MENS, AND M. CLAES, *On the topology of package dependency networks: A comparison of three programming language ecosystems*, in Proccedings of the 10th European Conference on Software Architecture Workshops, ACM, 2016, p. 21.
6.1

[75] A. DECAN, T. MENS, AND E. CONSTANTINOU, *On the evolution of technical lag in the npm package dependency network*, in Proceedings of the 34th International Conference on Software Maintenance and Evolution, IEEE, 2018, pp. 404–414.
2.2.2

[76] ——, *On the impact of security vulnerabilities in the npm package dependency network*, in Proceedings of the 15th International Conference on Mining Software Repositories, IEEE/ACM, 2018, pp. 181–191.

[77] A. DECAN, T. MENS, AND P. GROSJEAN, *An empirical comparison of dependency network evolution in seven software packaging ecosystems*, Empirical Software Engineering, 24 (2019), pp. 381–416.

2.2.2, 5.1, 6.1, 6.4.1, 13

[78] M. DÉNÈS, E. TASSI, ET AL., *merge-pr.sh*.
`https://github.com/coq/coq/blob/876f767/dev/tools/merge-pr.sh`, 2017–2019.
Last accessed October 9th, 2019.
3.7

[79] L. F. DIAS, I. STEINMACHER, G. PINTO, D. A. DA COSTA, AND M. GEROSA, *How does the shift to GitHub impact project collaboration?*, in Proceedings of the 32nd International Conference on Software Maintenance and Evolution, IEEE, 2016, pp. 473–477.
3.1, 3.2.2, 3.8

[80] J. DIETRICH, D. J. PEARCE, J. STRINGER, A. TAHIR, AND K. BLINCOE, *Dependency versioning in the wild*, in Proceedings of the 16th International Conference on Mining Software Repositories, IEEE, 2019, pp. 349–359.
2.2.2, 6.1

[81] J. DIMINO, R. GRINBERG, ET AL., *Dune: A composable build system for OCaml*.
`https://dune.build`, 2016–2019.
Last accessed September 7th, 2019.
2.5.1, 4.4.5.2, 7.3.2

[82] F. DOGLIO, *The latest npm breach... or is it?*
`https://blog.logrocket.com/the-latest-npm-breach-or-is-it-a427617a4185/`, 2018.
Last accessed July 23rd, 2019.
3

[83] E. DOLSTRA, M. DE JONGE, E. VISSER, ET AL., *Nix: A safe and policy-free system for software deployment*, in Large Installation Systems Administration conference, vol. 4, 2004, pp. 79–92.
2.4.1, 2.5.1, 6.3.1, 6.3.2

[84] K. DOMEN, *Cachix: Build Nix packages once and share them for good*.
`https://cachix.org/`, 2018–2019.
Last accessed September 18th, 2019.
6.5.2

[85] E. DURÁN, *CocoaPods trunk*.
`http://blog.cocoapods.org/CocoaPods-Trunk/`, 2014.
Last accessed August 30th, 2019.
6.4.1, 6.4.2

[86] J. EDGE, *Machine learning and stable kernels*.
`https://lwn.net/Articles/764647/`, 2018.
Last accessed September 16th, 2019.
5.4.1

[87] N. EGHBAL, B. KEEPERS, S. WILLS, M. LINKSVAYER, ET AL., *How to contribute to open source*.
`https://opensource.guide/how-to-contribute/`, 2016–2019.
Last accessed September 9th, 2019.
3.1

[88] O. ELAZHARY, M.-A. STOREY, N. ERNST, AND A. ZAIDMAN, *Do as I do, not as I say: Do contribution guidelines match the GitHub contribution process?*, in 35th International Conference on Software Maintenance and Evolution, IEEE, 2019.
3.4

[89] J. R. ERENKRANTZ, *Release management within open source projects*, in Proceedings of the 3rd Workshop on Open Source Software Engineering, 2003.
5.1

[90] M. EWING, E. TROAN, ET AL., *RPM package manager*.
`https://rpm.org/`, 1997–2018.
Last accessed September 17th, 2019.
6.3.1

[91] FACEBOOK, INC., *GraphQL*.
`https://graphql.github.io/graphql-spec/June2018/`, 2015–2018.
Last accessed September 9th, 2019.
2.5.2

[92] ——, *OCamlFormat*.
`https://github.com/ocaml-ppx/ocamlformat`, 2017–2019.
Last accessed October 6th, 2019.
2.5.1

[93] M. FERREIRA, T. MOMBACH, M. T. VALENTE, AND K. FERREIRA, *Algorithms for estimating truck factors: a comparative study*, Software Quality Journal, (2019), pp. 1–35.
7.2.2.1

[94] R. T. FIELDING, *Architectural Styles and the Design of Network-based Software Architectures*, PhD thesis, University of California at Irvine, 2000.
2.2.1, 6.4.1

[95] K. FOGEL, *Producing open source software: How to run a successful free software project*, O'Reilly Media, Inc., 2005.
1, 5.1, 5.2, 5.3.1, 7.3.3.1, 8.2.4

[96] C. FRANCALANCI AND F. MERLO, *Empirical analysis of the bug fixing process in open source projects*, in Proceedings of the 4th International Conference on Open Source Systems, IFIP, 2008, pp. 187–196.
4.2.2

[97] S. FRANKLIN AND A. GRAESSER, *Is it an agent, or just a program? A taxonomy for autonomous agents*, in International Workshop on Agent Theories, Architectures, and Languages, Springer, 1996, pp. 21–35.
2.5

[98] FREE SOFTWARE FOUNDATION, *Change logs*.
`https://www.gnu.org/prep/standards/html_node/Change-Logs.html`, 1992–2019.
In GNU Coding Standards. Last accessed August 26th, 2019.
5.5.1

[99] S. FUCHS ET AL., *Travis CI*.
`https://travis-ci.com/`, 2011–2019.
Last accessed September 9th, 2019.
2, 3.6.1, 3.6.4, 5.1

[100] I. GALIĆ, *Building Puppet's unofficial forge community*.
`https://opensource.com/article/17/6/vox-pupuli`, 2017.
Last accessed September 6th, 2019.
7.4.4

[101] E. GALLEGO ARIAS, T. ZIMMERMANN, ET AL., *Information for external library / Coq plugin authors*.
`https://github.com/coq/coq/blob/7282163/dev/ci/README-users.md`, 2017–2019.
Last accessed September 10th, 2019.
3.6.3.2

[102] E. GALLEGO ARIAS, T. ZIMMERMANN, G. GILBERT, ET AL., *Information for developers about the CI system*.
`https://github.com/coq/coq/blob/ae5ffa2/dev/ci/README-developers.md`, 2017–2019.
Last accessed September 11th, 2019.
3.6.6.3

[103] J. GAMALIELSSON AND B. LUNDELL, *Sustainability of open source software communities beyond a fork: How and why has the LibreOffice project evolved?*, Journal of Systems and Software, 89 (2014), pp. 128–145.
7.3.3.1

[104] T. GAMBLIN, M. LEGENDRE, M. R. COLLETTE, G. L. LEE, A. MOODY, B. R. DE SUPINSKI, AND S. FUTRAL, *The Spack package manager: bringing order to HPC software chaos*, in International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2015.
6.3.1

[105] A. GAWANDE, *Checklist manifesto*, Penguin Books India, 2010.
3.5.2, 5.6

[106] T. GAZAGNAIRE, L. GESBERT, ET AL., *The opam manual*.
`http://opam.ocaml.org/doc/Manual.html`, 2013–2019.
Last accessed September 17th, 2019.
6.4.1

[107] ——, *opam: OCaml package manager*.
`https://opam.ocaml.org/`, 2013–2019.
Last accessed September 10th, 2019.
3.6.3.2, 1, 6.5.1

[108] GCC DEVELOPMENT TEAM, *GCC development plan*.
`https://www.gnu.org/software/gcc/develop.html`.
Last accessed August 24th, 2019.
5.1, 5.3.1

[109] A. GELMAN AND G. IMBENS, *Why high-order polynomials should not be used in regression discontinuity designs*, Journal of Business & Economic Statistics, 37 (2018), pp. 1–10.
2.3.2.2

[110] L. GESBERT ET AL., *opam-publish*.
`https://github.com/ocaml/opam-publish`, 2014–2016.
Last accessed September 17th, 2019.
6.4.1

[111] GITHUB, *Assigning issues and pull requests to other GitHub users*.
https://help.github.com/en/articles/assigning-issues-and-pull-requests-to-other-github-users.
Last accessed September 3rd, 2019.
7.2

[112] ——, *GitHub REST API v3, GitHub developer guide.*
`https://developer.github.com/v3/`.
Last accessed August 15th, 2019.
2.2.1, 2.5.1, 4.3.2

[113] ——, *Searching in forks.*
`https://help.github.com/en/articles/searching-in-forks`.
Last accessed September 3rd, 2019.
7.3.3.2

[114] ——, *Force push timeline event.*
`https://github.blog/changelog/2018-11-15-force-push-timeline-event/`, 2018.
Last accessed August 22th, 2019.
30

[115] ——, *Related issues (public beta).*
`https://github.blog/changelog/2018-11-05-related-issues/`, 2018.
Last accessed September 13th, 2019.
4.4.5.4

[116] ——, *Apps.*
`https://developer.github.com/apps/`, Last accessed August 16th, 2019.
2.5.4

[117] ——, *GitHub GraphQL API v4, GitHub developer guide.*
`https://developer.github.com/v4/`, Last accessed June 26th, 2019.
2.2.1, 2.5.1, 4.4.2.1

[118] GitLab B.V., *GitLab.*
`https://about.gitlab.com/`, 2011–2019.
Last accessed September 9th, 2019.
2.5, 3.1, 3.6.1, 3.6.4, 5.1

[119] ——, *GraphQL API.*
`https://docs.gitlab.com/ee/api/graphql/reference/index.html`, Last accessed August 15th, 2019.
2.5.1

[120] R. Gloster, *RFC 36: RFC process team amendment.*
`https://github.com/NixOS/rfcs/blob/master/rfcs/0036-rfc-process-team-amendment.md`, 2018.
Last accessed October 11th, 2019.
8.2.3

[121] G. GOUSIOS AND D. SPINELLIS, *GHTorrent: GitHub's data from a firehose*, in Proceedings of the 9th Working Conference on Mining Software Repositories, IEEE, 2012, pp. 12–21.
7.2.2.1

[122] G. GOUSIOS, M.-A. STOREY, AND A. BACCHELLI, *Work practices and challenges in pull-based development: the contributor's perspective*, in Proceedings of the 38th International Conference on Software Engineering, IEEE/ACM, 2016, pp. 285–296.
3.2.1, 3.4, 3.6.1

[123] G. GOUSIOS, A. ZAIDMAN, M.-A. STOREY, AND A. VAN DEURSEN, *Work practices and challenges in pull-based development: the integrator's perspective*, in Proceedings of the 37th International Conference on Software Engineering, vol. 1, IEEE, 2015, pp. 358–368.
3.2.1

[124] G. D. GREENWADE, *The Comprehensive TeX Archive Network*, TUGboat — Proceedings of the 1993 Annual Meeting of the TeX Users Group, 14 (1993).
6.3.2

[125] J. GROSS, *Coq bug minimizer*.
`https://github.com/JasonGross/coq-tools/blob/5f43e25/find-bug.py`, 2013–2018.
Last accessed September 11th, 2019.
3.6.6.2

[126] J. GROSS AND J.-C. LÉCHENET, *Python script for dealing with deprecated notations in Coq*.
`https://gist.github.com/JasonGross/9770653967de3679d131c59d42de6d17`, 2018.
Last accessed September 10th, 2019.
3.6.2

[127] HACKAGE TEAM, *Hackage*.
`https://hackage.haskell.org/`, 2005.
Last accessed September 20th, 2019.
6.3.2

[128] M. HALLIN, T. CICHOCINSKI, ET AL., *graphql_ppx_re: GraphQL ppx rewriter for Bucklescript/ReasonML written in ReasonML*.
`https://github.com/baransu/graphql_ppx_re`, 2018–2019.
Last accessed December 1st, 2019.
2.5.1

[129] Haskell Platform Infrastructure Team, *Haskell platform*.
http://www.haskell.org/platform/, 2014–2019.
Last accessed September 20th, 2019.
6.5.2

[130] C. Hausman and D. S. Rapson, *Regression discontinuity in time: Considerations for empirical applications*, Annual Review of Resource Economics, 10 (2018), pp. 533–552.
2.3.2.1, 4.4.6.1

[131] J. Hietaniemi, E. Ashton, A. Bjørn Hansen, and L. Lapworth, *CPAN frequently asked questions*.
https://www.cpan.org/misc/cpan-faq.html, 1998–2011.
Last accessed on September 3rd, 2019.
7.3.3.2, 7.3.3.2

[132] K. Hightower, *What's new in Go 1.6 — vendoring*, ; login:, 41 (2016).
7.3.2

[133] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, *Usage, costs, and benefits of continuous integration in open-source projects*, in Proceedings of the 31st International Conference on Automated Software Engineering, IEEE/ACM, 2016, pp. 426–437.
3.6.1

[134] T. Hoang, J. Lawall, R. J. Oentaryo, Y. Tian, and D. Lo, *PatchNet: a tool for deep patch classification*, in Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, IEEE, 2019, pp. 83–86.
5.4.1

[135] D. Hollinger, *Vox Pupuli — the importance of working together*.
https://www.youtube.com/watch?v=28izTNK_-00, 2017.
Talk at DevOpsDays KC, video last accessed September 6th, 2019.
7.4.4

[136] A. Hora, R. Robbes, M. T. Valente, N. Anquetil, A. Etien, and S. Ducasse, *How do developers react to API evolution? A large-scale empirical study*, Software Quality Journal, 26 (2018), pp. 161–191.
3.6.2

[137] K. Hornik, *The Comprehensive R Archive Network*, Wiley Interdisciplinary Reviews: Computational Statistics, 4 (2012), pp. 394–398.
6.3.2

[138] M. HOWELL ET AL., *Homebrew*.
http://brew.sh/, 2009–2019.
Last accessed September 17th, 2019.
6.3.1

[139] J. HOYT AND I. ŽUŽAK, *GitHub API for importing issues*.
https://gist.github.com/jonmagic/5282384165e0f86ef105, 2015–2016.
Last accessed September 12th, 2019.
4.3.2

[140] J. HUA, *A case study of cross-branch porting in Linux kernel*, Master thesis, University of Texas at Austin, 2014.
5.3.2, 5.4.1

[141] J. HUANG AND M. CAKMAK, *Supporting mental model accuracy in trigger-action programming*, in Proceedings of the 2015 International Joint Conference on Pervasive and Ubiquitous Computing, ACM, 2015, pp. 215–225.
2.5.2

[142] J. D. HUNTER, *Matplotlib: A 2D graphics environment*, Computing in Science Engineering, 9 (2007), pp. 90–95.
2.4.2

[143] S. HYKES ET AL., *Docker*.
https://www.docker.com/, 2013–2019.
Last accessed September 18th, 2019.
6.5.2

[144] I. JACKSON, D. A. MORRIS, C. SHWARZ, C. LAMETER, J. GILBEY, ET AL., *Debian policy manual*.
https://www.debian.org/doc/debian-policy/index.html, 1996–2019.
Last accessed August 31th, 2019.
6.4.2

[145] R. JACOB, P. ZHU, M.-A. SOMERS, AND H. BLOOM, *A Practical Guide to Regression Discontinuity*, MDRC, 2012.
2.3.2.1

[146] M. JAMBON ET AL., *Yojson: Low-level JSON parsing and pretty-printing library for OCaml*.
https://ocaml-community.github.io/yojson/yojson/Yojson/index.html, 2012–2019.
Last accessed September 7th, 2019.
2.5.1

[147] JANE STREET GROUP, *JaneStreet's Base Library: Standard library for OCaml*. `https://opensource.janestreet.com/base/`, 2019.
Last accessed September 8th, 2019.
2.5.1

[148] S. JANSEN, *Measuring the health of open source software ecosystems: Beyond the scope of project health*, Information and Software Technology, 56 (2014), pp. 1508–1519.
7.2.2.3

[149] C. JASPAN, M. JORDE, A. KNIGHT, C. SADOWSKI, E. K. SMITH, C. WINTER, AND E. MURPHY-HILL, *Advantages and disadvantages of a monolithic repository: a case study at Google*, in Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ACM, 2018, pp. 225–234.
3.6.5

[150] C. JENSEN AND W. SCACCHI, *Role migration and advancement processes in OSSD projects: A comparative case study*, in Proceedings of the 29th International Conference on Software Engineering, IEEE/ACM, 2007, pp. 364–374.
3.4, 4.1

[151] E. JONES, T. OLIPHANT, P. PETERSON, ET AL., *SciPy: Open source scientific tools for Python*, 2001.
Last accessed August 17th, 2019.
2.4.2

[152] N. JØRGENSEN, *Putting it all in the trunk: Incremental software development in the FreeBSD open source project*, Information Systems Journal, 11 (2001), pp. 321–336.
5.1

[153] P. JUPYTER, M. BUSSONNIER, J. FORDE, J. FREEMAN, B. GRANGER, T. HEAD, C. HOLDGRAF, K. KELLEY, G. NALVARTE, A. OSHEROFF, ET AL., *Binder 2.0 — reproducible, interactive, sharable environments for science at scale*, in Proceedings of the 17th Python in Science Conference, 2018, pp. 113–120.
2.4.1

[154] S. JUST, R. PREMRAJ, AND THOMAS ZIMMERMANN, *Towards the next generation of bug tracking systems*, in Proceedings of the 2008 Symposium on Visual Languages and Human-Centric Computing, IEEE, 2008, pp. 82–85.
4.2.1

[155] B. KAHLE AND B. GILLIAT, *Wayback machine*, 2001.
2

[156] J.-O. KAISER, B. ZILIANI, R. KREBBERS, Y. RÉGIS-GIANAS, AND D. DREYER, *Mtac2: typed tactics for backward reasoning in Coq*, Proceedings of the ACM on Programming Languages, 2 (2018), p. 78.
1.3

[157] E. KALLIAMVAKOU, G. GOUSIOS, K. BLINCOE, L. SINGER, D. M. GERMAN, AND D. DAMIAN, *An in-depth study of the promises and perils of mining GitHub*, Empirical Software Engineering, 21 (2016), pp. 2035–2071.
3.5.1, 7.2.2.2

[158] R. KALLIS, A. DI SORBO, G. CANFORA, AND S. PANICHELLA, *Ticket tagger: Machine learning driven issue classification*, in Proceedings of the 35th International Conference on Software Maintenance and Evolution, IEEE, 2019.
8.2.2

[159] S. A. KARRE, A. SHUKLA, AND Y. R. REDDY, *Does your bug tracking tool suit your needs? A study on open source bug tracking tools*, arXiv preprint arXiv:1706.06799, (2017).
4.2.1

[160] A. KATILIUS, *Improving code quality using pull requests*.
`https://katilius.dev/improving-code-quality-using-pull-request/`, 2019.
Last accessed September 9th, 2019.
3.5

[161] K. KAWAGUCHI ET AL., *Jenkins*.
`https://jenkins.io/`, 2011–2019.
Last accessed September 9th, 2019.
2, 3.6.1

[162] B. KEEPERS, *Stale probot*.
`https://probot.github.io/apps/stale/`, 2017–2019.
Last accessed October 11th, 2019.
2

[163] N. KERZAZI AND P. N. ROBILLARD, *Kanbanize the release engineering process*, in 1st International Workshop on Release Engineering (RELENG), IEEE, 2013, pp. 9–12.
5.4.1

[164] F. KHOMH, T. DHALIWAL, Y. ZOU, AND B. ADAMS, *Do faster releases improve software quality? An empirical case study of Mozilla Firefox*, in Proceedings of the 9th Working Conference on Mining Software Repositories, IEEE, 2012, pp. 179–188.
5.1

[165] J. KHONDHU, A. CAPILUPPI, AND K.-J. STOL, *Is it all lost? A study of inactive open source projects*, in International Conference on Open Source Systems, IFIP, 2013, pp. 61–79.
7.2.2.2, 10

[166] R. KIKAS, G. GOUSIOS, M. DUMAS, AND D. PFAHL, *Structure and evolution of package dependency networks*, in Proceedings of the 14th International Conference on Mining Software Repositories, IEEE, 2017, pp. 102–112.
6.1

[167] B. A. KITCHENHAM, T. DYBA, AND M. JORGENSEN, *Evidence-based software engineering*, in Proceedings of the 26th International Conference on Software Engineering, IEEE, 2004, pp. 273–281.
2.3.1

[168] T. KLUYVER, B. RAGAN-KELLEY, F. PÉREZ, B. GRANGER, M. BUSSONNIER, J. FREDERIC, K. KELLEY, J. HAMRICK, J. GROUT, S. CORLAY, P. IVANOV, D. AVILA, S. ABDALLA, AND C. WILLING, *Jupyter notebooks: a publishing format for reproducible computational workflows*, in Positioning and Power in Academic Publishing: Players, Agents and Agendas, F. Loizides and B. Schmidt, eds., IOS Press, 2016, pp. 87 – 90.
2.4.1

[169] P. KOROLEV, *GitHub changelog generator*.
`https://github.com/github-changelog-generator/github-changelog-generator`, 2016–2019.
Last accessed August 26th, 2019.
5.5.3

[170] G. KROAH-HARTMAN, *The Linux kernel driver interface ("stable API nonsense")*.
`https://www.kernel.org/doc/html/v4.11/process/stable-api-nonsense.html`, 2004.
Last accessed August 19th, 2019.
3.6.3.1

[171] O. LACAN, *Keep a changelog*.
`https://keepachangelog.com/en/1.0.0/`, 2017.
Last accessed August 26th, 2019.
5.1, 5.5.1, 5.5.3

[172] J. LAMBERT, *GitLab 12.2 released with directed acyclic graphs for pipelines and design management*.
`https://about.gitlab.com/2019/08/22/gitlab-12-2-released/`, 2019.
Last accessed August 23rd, 2019.

3.6.4

[173] D. S. LEE AND T. LEMIEUX, *Regression discontinuity designs in economics*, Journal of economic literature, 48 (2010), pp. 281–355.
2.3.2.1

[174] M. M. LEHMAN, *Programs, life cycles, and laws of software evolution*, Proceedings of the IEEE, 68 (1980), pp. 1060–1076.
1

[175] M. M. LEHMAN, *Laws of software evolution revisited*, in European Workshop on Software Process Technology, Springer, 1996, pp. 108–124.
5.1, 1

[176] X. LEROY, D. DOLIGEZ, A. FRISCH, J. GARRIGUE, D. RÉMY, AND J. VOUILLON, *The OCaml system release 4.07: Documentation and user's manual*, tech. rep., Inria, July 2018.
2.4.2, 2.5.1, 4.4.5.2

[177] P. LETOUZEY ET AL., *Migrating the Coq repository from SVN to git*.
`https://github.com/coq/coq/wiki/MigrationGit`, 2013.
Last accessed August 17th, 2019.
3.3

[178] B. P. LIENTZ AND E. B. SWANSON, *Problems in application software maintenance*, Communications of the ACM, 24 (1981), pp. 763–769.
2

[179] J. LINDENBAUM, A. WIGGINS, AND O. HENRY, *Heroku: Cloud application platform*.
`https://heroku.com/`, 2007–2019.
Last accessed October 6th, 2019.
2.5.1

[180] W. MA, L. CHEN, X. ZHANG, Y. ZHOU, AND B. XU, *How do developers fix cross-project correlated bugs? A case study on the GitHub scientific Python ecosystem*, in Proceedings of the 39th International Conference on Software Engineering, IEEE/ACM, 2017, pp. 381–392.
3.1, 3.6.2, 4.4.5.2

[181] D. MACKENZIE, B. ELLISTON, P. EGGERT, E. BLAKE, ET AL., *Autoconf*.
`https://www.gnu.org/software/autoconf/`, 1996–2017.
Last accessed September 17th, 2019.
6.2

[182] A. Madhavapeddy, S. Zacchiroli, T. G. David Sheets, D. Scott, R. Grinberg, and A. Ray, *Cohttp: An OCaml library for HTTP clients and servers*.
`https://github.com/mirage/ocaml-cohttp`, 2009–2019.
Last accessed September 7th, 2019.
2.5.1

[183] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, *Managing the complexity of large free and open source package-based software distributions*, in Proceedings of the 21st International Conference on Automated Software Engineering, IEEE/ACM, 2006, pp. 199–208.
6.2, 6.4.4

[184] K. Manheimer, B. A. Warsay, S. N. Clark, and W. Rowe, *Depot: A framework for sharing software installation across organizational and Unix platform boundaries*, in Large Installation Systems Administration conference, USENIX, 1990, pp. 37–46.
6.3.1

[185] E. Martin-Dorel, *How to add overlays for PRs*.
`https://github.com/math-comp/math-comp/wiki/How-to-add-overlays-for-PRs`, 2019.
Last accessed September 10th, 2019.
3.6.4

[186] N. D. Matsakis and F. S. Klock II, *The Rust language*, in ACM SIGAda Ada Letters, vol. 34, ACM, 2014, pp. 103–104.
1.2

[187] T. Maurer, *Protected branches improvements*.
`https://github.blog/2016-03-30-protected-branches-improvements/`, 2016.
Last accessed August 23rd, 2019.
3.7

[188] W. McKinney, *Data structures for statistical computing in Python*, in Proceedings of the 9th Python in Science Conference, S. van der Walt and J. Millman, eds., 2010, pp. 51–56.
2.4.2

[189] K. McMinn, *New checks API public beta*.
`https://developer.github.com/changes/2018-05-07-new-checks-api-public-beta/`, 2018.
Last accessed August 16th, 2019.
2.5.4

[190] MELPA TEAM, *Contributing a new recipe to MELPA*.
`https://github.com/melpa/melpa/blob/master/CONTRIBUTING.org`.
Last accessed September 3rd, 2019.
7.3.3.2

[191] T. MENS, *An ecosystemic and socio-technical view on software maintenance and evolution*, in Proceedings of the 34th International Conference on Software Maintenance and Evolution, IEEE, 2016.
6.1

[192] T. MENS, B. ADAMS, AND J. MARSAN, *Towards an interdisciplinary, socio-technical analysis of software ecosystem health*, arXiv preprint arXiv:1711.04532, (2017).
7.2.2.3

[193] B. MEYER, *Empirical answers to important software engineering questions*.
https://cacm.acm.org/blogs/blog-cacm/224351-empirical-answers-to-important-software-engineering-questions-part-1-of-2/fulltext, 2018.
BLOG@CACM. Last accessed September 26th, 2019.
1.4

[194] L. A. MEYEROVICH AND A. S. RABKIN, *Empirical analysis of programming language adoption*, in ACM SIGPLAN Notices, vol. 48, ACM, 2013, pp. 1–18.
6.1

[195] M. MICHLMAYR, *Quality Improvement in Volunteer Free and Open Source Software Projects: Exploring the Impact of Release Management*, PhD thesis, University of Cambridge, 2007.
5.1

[196] MICROSOFT, *Azure pipelines*.
`https://azure.microsoft.com/en-us/services/devops/pipelines/`.
Last accessed September 9th, 2019.
3.6.1, 5.1

[197] T. MOMBACH AND M. T. VALENTE, *GitHub REST API vs GHTorrent vs GitHub Archive: A comparative study*.
`https://homepages.dcc.ufmg.br/~mtov/pub/2018-vem-thais.pdf`, 2018.
Last accessed September 8th, 2019.
2.2.1

[198] J. T. MORGAN AND A. HALFAKER, *Evaluating the impact of the Wikipedia Teahouse on newcomer socialization and retention.*, in Proceedings of the 14th International Symposium on Open Collaboration (OpenSym), ACM, 2018.

2

[199] I. MURDOCK ET AL., *Dpkg*.
https://git.dpkg.org/git/dpkg/dpkg.git, 1994–2019.
Last accessed September 17th, 2019.
6.3.1

[200] E. MURPHY-HILL AND D. GROSSMAN, *How programming languages will co-evolve with software engineering: a bright decade ahead*, in Proceedings of the 36th International Conference on Software Engineering: Future of Software Engineering, ACM, 2014, pp. 145–154.
1.2

[201] K. MUŞLU, C. BIRD, N. NAGAPPAN, AND J. CZERWONKA, *Transition from centralized to decentralized version control systems: A case study on reasons, barriers, and outcomes*, in Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 334–344.
3.2.2

[202] K. NAKAKOJI, Y. YAMAMOTO, Y. NISHINAKA, K. KISHIDA, AND Y. YE, *Evolution patterns of open-source software systems and communities*, in Proceedings of the International Workshop on Principles of Software Evolution, ACM, 2002, pp. 76–85.
3.4, 4.1

[203] M. NASSIF AND M. P. ROBILLARD, *Revisiting turnover-induced knowledge loss in software projects*, in Proceedings of the 33th International Conference on Software Maintenance and Evolution, IEEE, 2017, pp. 261–272.
7.2.2.1

[204] R. NEHMER, *Keep your project boards up to date, automatically*.
https://github.blog/2017-10-30-keep-your-project-boards-up-to-date-automatically/, 2017.
Last accessed August 25th, 2019.
5.4.2

[205] A. NESBITT, *Package manager categories*.
https://github.com/ipfs/package-managers/blob/3f4a51e/docs/categories.md, 2019.
Last accessed September 17th, 2019.
6.4.1

[206] A. NESBITT AND B. NICKOLLS, *Libraries.io open source repository and dependency meta-data*, June 2017.
2.2.2, 6.4.1, 7.2

[207] A. Nesbitt and T. Zimmermann, *Academic papers related to package management and package ecosystems*.
https://github.com/ipfs/package-managers/blob/a09fb7a/docs/papers.md, 2019.
Last accessed September 17th, 2019.
6.1

[208] J. B. Nils Adermann et al., *Composer: A dependency manager for php*.
https://getcomposer.org/, 2012–2019.
Last accessed September 20th, 2019.
6.3.2

[209] npm, Inc., *npm-unpublish: Remove a package from the registry*.
https://docs.npmjs.com/cli/unpublish.
Last accessed August 30th, 2019.
6.4.1

[210] L. Nyman and J. Lindman, *Code forking, governance, and sustainability in open source software*, Technology Innovation Management Review, 3 (2013).
7.3.3.1

[211] L. Nyman and T. Mikkonen, *To fork or not to fork: Fork motivations in SourceForge projects*, International Journal of Open Source Software and Processes, 3 (2011), pp. 1–9.
7.3.3.1

[212] OCaml Labs, *OCaml platform*.
https://ocaml.org/platform/, 2019.
Last accessed September 20th, 2019.
6.5.2

[213] T. Oliphant, *NumPy: A guide to NumPy*.
USA: Trelgol Publishing, 2006.
Last accessed August 17th, 2019.
2.4.2

[214] R. Olson, *Release your software*.
https://github.blog/2013-07-02-release-your-software/, 2013.
Last accessed August 26th, 2019.
5.1, 5.5.1

[215] J. Ooms, *Possible directions for improving dependency versioning in R*, arXiv preprint arXiv:1303.2140, (2013).
6.1

[216] A. D. OXMAN, *Systematic reviews: checklists for review articles*, Bmj, 309 (1994), pp. 648–651.
3.5.2, 5.6

[217] J. PACE, *Introducing code owners*.
`https://github.blog/2017-07-06-introducing-code-owners/`, 2017.
Last accessed August 23rd, 2019.
3.1, 3.7

[218] K. PALMSKOG, A. CELIK, AND M. GLIGORIC, *piCoq: Parallel regression proving for large-scale verification projects*, in Proceedings of the 27th International Symposium on Software Testing and Analysis, ACM, 2018, pp. 344–355.
3.6.6.2

[219] T. H. PARKER, S. C. GRIFFITH, J. L. BRONSTEIN, F. FIDLER, S. FOSTER, H. FRASER, W. FORSTMEIER, J. GUREVITCH, J. KORICHEVA, R. SEPPELT, ET AL., *Empowering peer reviewers with a checklist to improve transparency*, Nature ecology & evolution, 2 (2018), p. 929.
3.5.2, 5.6

[220] S. PASAT, *Introducing GitHub package registry*.
`https://github.blog/2019-05-10-introducing-github-package-registry/`, 2019.
Last accessed August 27th, 2019.
2, 6.4.1

[221] ——, *Updates to GitHub package registry*.
`https://github.blog/2019-06-19-updates-to-github-package-registry/`, 2019.
Last accessed August 30th, 2019.
6.4.1

[222] PEAR GROUP, *PEAR: PHP extension and application repository*.
`https://pear.php.net/`, 1999.
Last accessed September 20th, 2019.
6.3.2, 7.4.5

[223] PEAR QUALITY ASSURANCE TEAM, *Taking over an unmaintained package*.
`https://pear.php.net/manual/en/newmaint.takingover.php`.
Last accessed September 4th, 2019.
7.3.3.2, 7.4.5

[224] F. PEREZ, B. E. GRANGER, AND J. D. HUNTER, *Python: an ecosystem for scientific computing*, Computing in Science & Engineering, 13 (2010), pp. 13–21.
2.4.2

[225] J. Perktold, S. Seabold, and J. Taylor, *StatsModels: Statistics in Python*.
https://www.statsmodels.org/stable/index.html, 2006–2019.
Last accessed August 17th, 2019.
2.4.2

[226] R. Pham, L. Singer, and K. Schneider, *Building test suites in social coding sites by leveraging drive-by commits*, in Proceedings of the 35th International Conference on Software Engineering, IEEE, 2013, pp. 1209–1212.
3.3

[227] Pip developers, *pip*.
https://pypi.org/project/pip/, 2011.
Last accessed September 20th, 2019.
6.3.2

[228] C. Pit-Claudel, *An experiment in porting Coq's manual to reStructuredText*.
https://github.com/cpitclaudel/coq-rst, 2016.
Last accessed August 17th, 2019.
1

[229] C. Pit-Claudel and P. Courtieu, *Company-Coq: Taking Proof General one step closer to a real IDE*, in 2nd International Workshop on Coq for PL, St. Petersburg, Florida, United States, Jan. 2016.
4.4.5.2

[230] N. Plasterer, *Requesting reviews from repository teams*.
https://github.blog/2017-06-28-requesting-reviews-from-repository-teams/, 2017.
Last accessed August 23rd, 2019.
3.7

[231] A. Popp et al., *esy: Easy package management for native Reason, OCaml and more*.
http://esy.sh/, 2018–2019.
Last accessed October 7th, 2019.
1

[232] T. Preston-Werner, *Semantic versioning 2.0.0*.
https://semver.org, 2013.
Last accessed August 24th, 2019.
5.1, 5.2, 5.6, 6.4.2

[233] ——, *Tom's obvious, minimal language (TOML)*.

`https://github.com/toml-lang/toml`, 2013–2016.
Last accessed September 18th, 2019.
6.4.2

[234] PYTHON SOFTWARE FOUNDATION, *PyPI: the Python package index*.
`https://pypi.org/`, 2002–2019.
Last accessed September 20th, 2019.
6.3.2

[235] M. T. RAHMAN AND P. C. RIGBY, *Contrasting development and release stabilization work on the Linux kernel*, in International Workshop on Release Engineering, Citeseer, 2014.
9

[236] ——, *Release stabilization on Linux and Chrome*, IEEE Software, 32 (2015), pp. 81–88.
5.1, 5.6

[237] A. RAHMATI, E. FERNANDES, J. JUNG, AND A. PRAKASH, *IFTTT vs. Zapier: A comparative study of trigger-action programming frameworks*, arXiv preprint arXiv:1709.02788, (2017).
2.5.2

[238] S. RAHTZ ET AL., *TeX Live distribution*.
`http://www.tug.org/texlive/`, 1996–2019.
Last accessed September 20th, 2019.
6.5.2

[239] E. RAYMOND, *The cathedral and the bazaar*, Knowledge, Technology & Policy, 12 (1999), pp. 23–49.
4.1

[240] E. S. RAYMOND AND G. L. STEELE, *The new hacker's dictionary*, Mit Press, 1996.
7.3.3.1

[241] K. REITZ, *Requests: HTTP for humans*.
`https://2.python-requests.org/en/master/`, 2011–2019.
Last accessed August 14th, 2019.
2.4.2

[242] R. REYNOLDS ET AL., *Chocolatey: The package manager for windows*.
`https://chocolatey.org/`, 2011–2019.
Last accessed September 17th, 2019.
6.3.1

[243] P. C. Rigby, Y. C. Zhu, S. M. Donadelli, and A. Mockus, *Quantifying and mitigating turnover-induced knowledge loss: Case studies of Chrome and a project at Avaya*, in Proceedings of the 38th International Conference on Software Engineering, IEEE/ACM, 2016, pp. 1006–1016.
7.2.2.1

[244] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, and Z. Tatlock, *QED at large: A survey of engineering of formally verified software*, Foundations and Trends in Programming Languages, 5 (2019), pp. 102–281.
1.3, 8.1

[245] T. Ringer, N. Yazdani, J. Leo, and D. Grossman, *Adapting proof automation to adapt proofs*, in Proceedings of the 7th International Conference on Certified Programs and Proofs, ACM, 2018, pp. 115–129.
3.6.2

[246] R. Rivest, *S-expressions*, Internet draft, Network Working Group, 1997.
6.4.2

[247] G. Robles and J. M. González-Barahona, *A comprehensive study of software forks: Dates, reasons and outcomes*, in International Conference on Open Source Systems, IFIP, 2012, pp. 1–14.
7.3.3.1

[248] G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz, *Evolution of the core team of developers in libre software projects*, in Proceedings of the 6th International Working Conference on Mining Software Repositories, IEEE, 2009.
3.4, 4.4.2.3

[249] C. Rodríguez-Bustos and J. Aponte, *How distributed version control systems impact open source software projects*, in Proceedings of the 9th International Working Conference on Mining Software Repositories, IEEE, 2012, pp. 36–39.
3.1

[250] R. Roveda, F. A. Fontana, C. Raibulet, M. Zanoni, and F. Rampazzo, *Does the migration to GitHub relate to internal software quality?*, in Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering, ScitePress, 2017, pp. 293–300.
3.2.2

[251] Rust team, *The Rust RFC book*.
`https://rust-lang.github.io/rfcs/introduction.html`, 2014–2019.

Last accessed October 11th, 2019.
8.2.3

[252] ——, *The Cargo book*.
`https://doc.rust-lang.org/stable/cargo/`, 2017–2019.
Last accessed September 2nd, 2019.
15

[253] J. SAMETINGER, *Software Engineering with Reusable Components*, Springer-Verlag, Berlin, Heidelberg, 1997.
6.1

[254] A. SARKAR, P. C. RIGBY, AND B. BARTALOS, *Improving bug triaging with high confidence predictions at Ericsson*, in Proceedings of the 35th International Conference on Software Maintenance and Evolution, IEEE, 2019.
8.2.2

[255] SCALA PLATFORM COMMITTEE, *Scala platform*.
`https://scalacenter.github.io/platform/platform.html`, 2016.
Last accessed September 20th, 2019.
6.5.2

[256] I. Z. SCHLUETER ET AL., *npm*.
`https://www.npmjs.com/`, 2010–2019.
Last accessed September 20th, 2019.
1, 6.3.2

[257] J. SEGAL AND C. MORRIS, *Developing scientific software*, IEEE Software, 25 (2008), pp. 18–20.
1.3

[258] E. SOGGE HEGGEN, *Free hosting for open source v2*.
`https://blog.discourse.org/2018/11/free-hosting-for-open-source-v2/`, 2018.
Last accessed September 9th, 2019.
3.4.1

[259] S. K. SOWE, R. A. GHOSH, AND K. HAALAND, *A multi-repository approach to study the topology of open source bugs communities: Implications for software and code quality*, International Journal of Computer Theory and Engineering, 5 (2013), p. 138.
4.2.2

[260] SPDX WORKGROUP, *Software package data exchange (SPDX)*.
`https://spdx.org/`, 2010–2016.

Last accessed September 18th, 2019.
6.4.2

[261] R. SPEICHER, *How we solved GitLab's changelog conflict crisis*.
https://about.gitlab.com/2018/07/03/solving-gitlabs-changelog-conflict-crisis/, 2018.
Last accessed August 26th, 2019.
5.5.2, 5.5.3

[262] D. SPINELLIS, *Version control systems*, IEEE Software, 22 (2005), pp. 108–109.
3.1

[263] M. SQUIRE, *"Should we move to Stack Overflow?" Measuring the utility of social media for developer support*, in Proceedings of the 37th International Conference on Software Engineering, vol. 2, IEEE/ACM, 2015, pp. 219–228.
3.4.1, 4.2.3

[264] STACKAGE TEAM, *Stackage*.
https://www.stackage.org/, 2015.
Last accessed September 20th, 2019.
6.3.2

[265] R. M. STALLMAN, *EMACS the extensible, customizable self-documenting display editor*, ACM, 1981.
2.5.1

[266] STATACORP, *Stata statistical software: Release 16*, tech. rep., Statacorp LLC, 2019.
2.4.2

[267] I. STEINMACHER, M. A. GRACIOTTO SILVA, M. A. GEROSA, AND D. F. REDMILES, *A systematic literature review on the barriers faced by newcomers to open source software projects*, Information and Software Technology, 59 (2015), pp. 67–85.
3.1

[268] M.-A. STOREY AND A. ZAGALSKY, *Disrupting developer productivity one bot at a time*, in Proceedings of the 24th International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 928–931.
2.5

[269] J. D. STRATE AND P. A. LAPLANTE, *A literature review of research in software defect reporting*, IEEE Transactions on Reliability, 62 (2013), pp. 444–454.
4.2

[270] SYMFONY TEAM, *Proposing a change*.
https://symfony.com/doc/current/contributing/code/pull_requests.html.
Last accessed August 24th, 2019.
5.3.2

[271] D. A. TAMBURRI, P. KRUCHTEN, P. LAGO, AND H. VAN VLIET, *Social debt in software engineering: insights from industry*, Journal of Internet Services and Applications, 6 (2015), p. 10.
1.3

[272] C. TEYTON, J.-R. FALLERI, AND X. BLANC, *Mining library migration graphs*, in Proceedings of the 19th Working Conference on Reverse Engineering, IEEE, 2012, pp. 289–298.
7.3.1

[273] ——, *Automatic discovery of function mappings between similar libraries*, in Proceedings of the 20th Working Conference on Reverse Engineering, IEEE, 2013, pp. 192–201.
7.3.1

[274] S. TISUE, *Scala community build grows to 141 projects, 2.8 million lines of code*.
https://www.scala-lang.org/2018/01/16/community-build-growth.html, 2018.
Last accessed September 11th, 2019.
3.6.5

[275] M. TODD, *Task lists in GFM: Issues/pulls, comments*.
https://github.blog/2013-01-09-task-lists-in-gfm-issues-pulls-comments/, 2013.
Last accessed August 18th, 2019.
3.5.2

[276] M. TORCHIANO, F. RICCA, AND A. MARCHETTO, *Is my project's truck factor low? Theoretical and empirical considerations about the truck factor threshold*, in Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics, ACM, 2011, pp. 12–18.
7.2.2.1

[277] G. TORIKIAN, B. BLACK, B. SWINNERTON, C. SOMERVILLE, D. CELIS, AND K. DAIGLE, *The GitHub GraphQL API*.
https://github.blog/2016-09-14-the-github-graphql-api/, 2016.
Last accessed August 13th, 2019.
2.2.1

[278] L. TORVALDS, *Tech talk: Linus Torvalds on git*.

`https://www.youtube.com/watch?v=4XpnKHJAok8`, 2007.
Transcript available at
`https://git.wiki.kernel.org/index.php/LinusTalk200705Transcript`.
Last accessed September 8th, 2019.
3.1

[279] L. TORVALDS, J. HAMANO, ET AL., *git*.
`https://git-scm.com/`, 2005–2019.
Last accessed September 8th, 2019.
3.1, 3.3

[280] P. TOURANI, B. ADAMS, AND A. SEREBRENIK, *Code of conduct in open source projects*, in Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017, pp. 24–33.
3

[281] A. TRIDGELL AND D. SHEARER, *Jitterbug*.
`https://jitterbug.samba.org/`.
Last accessed September 12th, 2019.
4.3.1

[282] A. TROCKMAN, S. ZHOU, C. KÄSTNER, AND B. VASILESCU, *Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem*, in Proceedings of the 40th International Conference on Software Engineering, IEEE/ACM, 2018, pp. 511–522.
2.3.2.3

[283] TROUPE TECHNOLOGY LTD., *Gitter*.
`https://gitlab.com/gitlab-org/gitter/webapp`, 2014–2017.
Last accessed September 9th, 2019.
11, 3.6.4

[284] U. UPADHYAY, *Lovely forks*.
`https://github.com/musically-ut/lovely-forks`, 2015–2018.
Last accessed September 3rd, 2019.
7.3.3.2

[285] M. VALIEV, B. VASILESCU, AND J. HERBSLEB, *Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem*, in Proceedings of the 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 2018, pp. 644–655.
7.2.1, 7.2.2.2

[286] D. W. van Liere, *How shallow is a bug? Why open source communities shorten the repair time of software defects*, in Proceedings of the 30th International Conference on Information Systems, 2009, p. 195.
4.1

[287] G. Van Rossum et al., *Python programming language.*, in USENIX annual technical conference, vol. 41, 2007, p. 36.
2.4.2, 8.2.3

[288] B. Vasilescu, A. Serebrenik, P. Devanbu, and V. Filkov, *How social Q&A sites are changing knowledge sharing in open source software communities*, in Proceedings of the 17th Conference on Computer Supported Cooperative Work & social computing, ACM, 2014, pp. 342–354.
3.4.1, 4.2.3

[289] M. Venetucci Harvey, *Updates to protected branches*.
https://github.blog/2018-08-15-protected-branch-updates/, 2018.
Last accessed August 23rd, 2019.
3.7

[290] S. Vessels, *Label improvements: emoji, descriptions, and more*.
https://github.blog/2018-02-22-label-improvements-emoji-descriptions-and-more/, 2018.
Last accessed August 19th, 2019.
3.5.1

[291] S. Vidal et al., *Yellowdog updater modified*.
http://yum.baseurl.org/.
Last accessed September 17th, 2019.
6.3.1

[292] J. Vouillon, *Lwt: a cooperative thread library*, in Proceedings of the 2008 ACM SIGPLAN workshop on ML, ACM, 2008, pp. 3–12.
2.5.1

[293] Vox Pupuli team, *Migrating a module to Vox Pupuli*.
https://voxpupuli.org/docs/migrate_module/, 2016–2018.
Last accessed September 6th, 2019.
7.4.4

[294] A. K. Wagner, S. B. Soumerai, F. Zhang, and D. Ross-Degnan, *Segmented regression analysis of interrupted time series studies in medication use research*, Journal of clinical pharmacy and therapeutics, 27 (2002), pp. 299–309.
4.4.6.1

[295] C. WANSTRATH, *A whole new GitHub universe: announcing new tools, forums, and features*.
https://github.blog/2016-09-14-a-whole-new-github-universe-announcing-new-tools-forums-and-features/, 2016.
Last accessed August 25th, 2019.
5.4.1

[296] C. WANSTRATH, P. J. HYETT, T. PRESTON-WERNER, AND S. CHACON, *GitHub: social code hosting*.
`https://github.com`, 2008–2019.
Last accessed September 27th, 2019.
3.1

[297] B. WARSAW, J. HYLTON, D. GOODGER, AND N. COGHLAN, *PEP 1: PEP purpose and guidelines*.
`https://www.python.org/dev/peps/pep-0001/`, 2000.
Last accessed October 11th, 2019.
8.2.3

[298] D. WEBB, *Forking to Sous Chefs*.
`https://sous-chefs.org/docs/forking/`, 2017.
Last accessed September 7th, 2019.
7.4.4

[299] ——, *Transferring to Sous Chefs*.
`https://sous-chefs.org/docs/transferring/`, 2017.
Last accessed September 7th, 2019.
7.4.4

[300] S. WEBER, *The success of open source*, Harvard University Press, 2004.
7.3.3.1

[301] Y. WEN, J. CAO, AND S. CHENG, *PTracer: A Linux kernel patch trace bot*, arXiv preprint arXiv:1903.03610, (2019).
5.4.1

[302] B. WHITE ET AL., *Advanced packaging tool*.
`https://salsa.debian.org/apt-team/apt`, 1998–2019.
Last accessed September 17th, 2019.
6.3.1

[303] D. WIDDER, B. VASILESCU, M. HILTON, AND C. KÄSTNER, *I'm leaving you, Travis: a continuous integration breakup story*, in Proceedings of the 15th International Conference on Mining Software Repositories, IEEE/ACM, 2018, pp. 165–169.

3.6.1

[304] A. WILLIAMS, *Changes to npm's unpublish policy*.
`https://blog.npmjs.org/post/141905368000`, 2016.
Last accessed August 30th, 2019.
6.4.1

[305] J. M. WOOLDRIDGE, *Introductory econometrics: A modern approach*, Nelson Education, 2015.
2.3.2.2, 3

[306] V. WU, *GitLab 10.6 released with CI/CD for GitHub and deeper Kubernetes integration*.
`https://about.gitlab.com/2018/03/22/gitlab-10-6-released/`, 2018.
Last accessed August 23rd, 2019.
3.6.4

[307] Y. WU, J. KROPCZNYSKI, R. PRATES, AND J. M. CARROLL, *The rise of curation on GitHub*, in 3rd AAAI conference on human computation and crowdsourcing, 2015.
6.2

[308] K. YAMASHITA, S. MCINTOSH, Y. KAMEI, A. E. HASSAN, AND N. UBAYASHI, *Revisiting the applicability of the Pareto principle to core development teams in open source software projects*, in Proceedings of the 14th International Workshop on Principles of Software Evolution, ACM, 2015, pp. 46–55.
7.2.2.1

[309] YARN CONTRIBUTORS, *Yarn*.
`https://yarnpkg.com/`, 2016–2019.
Last accessed September 20th, 2019.
1, 6.3.2

[310] Y. YE AND K. KISHIDA, *Toward an understanding of the motivation of open source software developers*, in Proceedings of the 25th International Conference on Software Engineering, IEEE/ACM, 2003, pp. 419–429.
3.4, 4.1

[311] A. ZEROUALI, T. MENS, G. ROBLES, AND J. M. GONZALEZ-BARAHONA, *On the diversity of software package popularity metrics: An empirical study of npm*, in Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2019, pp. 589–593.
2.2.2

[312] T. Zhang, H. Jiang, X. Luo, and A. T. Chan, *A literature review of research in bug resolution: Tasks, challenges and future directions*, The Computer Journal, 59 (2016), pp. 741–773.
4.2

[313] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, *The impact of continuous integration on other software development practices: a large-scale empirical study*, in Proceedings of the 32nd International Conference on Automated Software Engineering, IEEE/ACM, 2017, pp. 60–71.
2.3.2.3

[314] S. Zhou, B. Vasilescu, and C. Kästner, *What the fork: a study of inefficient and efficient forking practices in social coding*, in Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 2019, pp. 350–361.
7.3.3.2, 7.4.6

[315] T. Zimmermann, *Getting thousands to millions of people working on a single mathematical proof*.
`https://www.theozimmermann.net/2014/11/06/collaborative-proofs/`, 2014.
Last accessed October 10th, 2019.
8.1

[316] ——, *A key to massively collaborative math is a good interface for writing proofs*.
`https://www.theozimmermann.net/2014/12/feedback-collaborative-proofs/`, 2014.
Last accessed October 10th, 2019.
8.1

[317] ——, *What good can massive-scale collaboration bring to science?*
`https://www.theozimmermann.net/2014/09/15/massive-collaboration-in-science/`, 2014.
Last accessed October 10th, 2019.
8.1

[318] ——, *Assessment of tests with external reverse dependencies*.
`https://mybinder.org/v2/gh/Zimmi48/thesis/master?filepath=notebooks%2Fbreaking-changes%2FBreaking_changes.ipynb`, 2019.
Last accessed October 13th, 2019.
1.5.3, 2.4.1, 3.6.4

[319] ——, *Automatic discovery of community organizations for long-term package maintenance*.

https://mybinder.org/v2/gh/Zimmi48/thesis/master?filepath=notebooks%2Fcommunity-orgs%2FCommunity_organizations.ipynb, 2019.
Last accessed October 13th, 2019.
7.4.4

[320] ——, *Empirical evaluation of the impact of labels and templates on pull request quality*.
https://mybinder.org/v2/gh/Zimmi48/thesis/master?filepath=notebooks%2Ftemplates%2FTemplates.ipynb, 2019.
Last accessed October 13th, 2019.
3.5.3, 3.5.3.5

[321] ——, *Evaluation of the prevalence of single-maintainer packages*.
https://mybinder.org/v2/gh/Zimmi48/thesis/master?filepath=notebooks%2Flibraries-io%2FLibraries.io.ipynb, 2019.
Last accessed October 13th, 2019.
7.2

[322] ——, *Evolution of the number of contributors over time*.
https://mybinder.org/v2/gh/Zimmi48/thesis/master?filepath=notebooks%2Fgithub-contributors%2FGitHub_contributors.ipynb, 2019.
Last accessed October 13th, 2019.
3.3

[323] T. ZIMMERMANN AND A. CASANUEVA ARTÍS, *Empirical evaluation of the switch of bug trackers*.
https://mybinder.org/v2/gh/Zimmi48/thesis/master?filepath=notebooks%2Fbug-tracker%2FGitHub_migration.ipynb, 2019.
Last accessed October 13th, 2019.
1.5.3, 2.4.1, 4.4.2.1, 4.4.4.5, 4.4.6.1

[324] ——, *Impact of switching bug trackers: a case study on a medium-sized open source project*, in Proceedings of the 35th International Conference on Software Maintenance and Evolution, IEEE, 2019.
1.5.4, 4.1

[325] ——, *Source and supporting data for "Impact of switching bug trackers: a case study on a medium-sized open source project"*.
https://github.com/Zimmi48/impact-of-switching-bug-trackers, 2019.
Last accessed January 10th, 2020.
4.4.5

[326] T. ZIMMERMANN ET AL., *Release process (checklist)*.

`https://github.com/coq/coq/blob/e6322e2/dev/doc/release-process.md`, 2018–2019.
Last accessed September 16th, 2019.
5.6

[327] T. ZIMMERMANN, E. GALLEGO ARIAS, ET AL., *Add overlays for your pull requests in this directory*.
`https://github.com/coq/coq/blob/df804ec/dev/ci/user-overlays/README.md`, 2017–2019.
Last accessed September 11th, 2019.
3.6.6.3

[328] W. ZOU, W. ZHANG, X. XIA, R. HOLMES, AND Z. CHEN, *Branch use in practice: A large-scale empirical study of 2,923 projects on GitHub*, in Proceedings of the 19th International Conference on Software Quality, Reliability, and Security, IEEE, 2019.
5.3.1

LaTeX thesis template adapted from Víctor F. Breña-Medina's template (Creative Commons CC BY 4.0)

`https://frama.link/thesis_template`