When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems

CHRIS BOGART, CHRISTIAN KÄSTNER, and JAMES HERBSLEB,
Carnegie Mellon University, USA
FERDIAN THUNG, Singapore Management University, Singapore

Open source software projects often rely on package management systems that help projects discover, incorporate, and maintain dependencies on other packages, maintained by other people. Such systems save a great deal of effort over ad hoc ways of advertising, packaging, and transmitting useful libraries, but coordination among project teams is still needed when one package makes a breaking change affecting other packages. Ecosystems differ in their approaches to breaking changes, and there is no general theory to explain the relationships between features, behavioral norms, ecosystem outcomes, and motivating values. We address this through two empirical studies. In an interview case study, we contrast Eclipse, NPM, and CRAN, demonstrating that these different norms for coordination of breaking changes shift the costs of using and maintaining the software among stakeholders, appropriate to each ecosystem's mission. In a second study, we combine a survey, repository mining, and document analysis to broaden and systematize these observations across 18 ecosystems. We find that all ecosystems share values such as stability and compatibility, but differ in other values. Ecosystems' practices often support their espoused values, but in surprisingly diverse ways. The data provides counterevidence against easy generalizations about why ecosystem communities do what they do.

CCS Concepts: • **Software and its engineering → Collaboration in software development**; **Software development process management**; **Software libraries and repositories**; • **Human-centered computing → Empirical studies in collaborative and social computing**;

Additional Key Words and Phrases: Software ecosystems, dependency management, semantic versioning, collaboration, qualitative research

ACM Reference format:
Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 42 (July 2021), 56 pages.
https://doi.org/10.1145/3447245

This work has been supported by by NSF awards 1901311, 1546393, 1302522, 1322278, 0943168, 1318808, 1633083, and 1552944, the Science of Security Lablet (H9823014C0140), the U.S. Department of Defense through the Systems Engineering Research Center, and a grant from the Alfred P. Sloan Foundation.

Authors' addresses: C. Bogart, C. Kästner, and J. Herbsleb, Carnegie Mellon University, Institute for Software Research TCS Hall 430, 4665 Forbes Avenue, Pittsburgh, PA 15213; emails: {cbogart, ckaestner, jherbsleb}@cs.cmu.edu; F. Thung, Singapore Management University, School of Computing and Information Systems, 80 Stamford Road, Singapore 178902; email: ferdiant.2013@smu.edu.sg.

1049-331X/2021/07-ART42 $15.00
https://doi.org/10.1145/3447245

1 INTRODUCTION

Software ecosystems are communities built around shared programming languages, shared platforms, or shared dependency management tools, which allow developers to create packages that import and build on each others' functionality. Software ecosystems have become an important paradigm for organizing open source software development and maintaining and reusing code packages. Development within ecosystems is efficient in the sense that common functionalities need only be developed, maintained, and tested by a single team, instead of many authors reimplementing the same functionality.

Coordination is a major challenge in software ecosystems, since packages tend to be highly interdependent yet independently maintained [2, 3, 6, 21, 55, 68]. In at least some ecosystems, such as JavaScript, transitive dependency networks are growing rapidly [46]. Improvements that a maintainer makes to a shared package may affect many users of that package, for example, by incorporating new features, making APIs simpler, and improving maintainability [10]. Any of these actions may require rework from developers whose software depends on that package. Package users may invest in regular rework to keep up with changes, collaborate with upstream projects to minimize the impact of those changes, decline to update to the latest versions (at the risk of missing bug fixes or security updates), or replicate functionality to avoid dependencies in the first place [6, 17, 19, 72]. Package maintainers, in turn, have many ways to reduce the burden on their users. For example, they can refrain from performing changes, announce and clearly label breaking changes, or help their users to migrate from old to new versions [6, 36, 65, 67]. Many different practices can contribute to managing change, and adopting various practices can shift some of the cost (in the form of effort) among different classes of ecosystem participants such as maintainers, package users, and end-users (e.g., Reference [28]) .

While much is known about some individual practices for managing change, we do not yet understand how these practices occur in the wild, nor how they can combine to establish the full design space of practices. Managing change takes time and effort from both upstream and downstream developers, and depending on their community's practices, this cost may be distributed differently. However, we do not fully understand the distributions of costs that result from various practices, nor how practices are related to ecosystem culture and technologies. This is important not only from a research perspective, to acquire an understanding of ecosystem coordination mechanisms, but also for practitioners and sponsors who may need to tune the distribution of costs to accommodate changing conditions. For example, as an ecosystem accumulates a large and rapidly growing base of applications that use particular packages, its community may wish to adopt practices to increase the stability of those packages to avoid imposing the costs of change on a large and growing base of users. What practices could accomplish this? Of this set of practices, which are likely to be compatible with the adopting ecosystem's culture and values?

We perform two studies to address questions like this. First, we conducted a multiple case study (Study 1) of three open source software ecosystems with different philosophies toward change: Eclipse, R/CRAN, and Node.js/npm. We studied how developers plan, manage, and coordinate change within each ecosystem, how change-related costs are allocated, and how developers are influenced by and influence change-related expectations, policies, and tools in the ecosystem. In each ecosystem, we studied public policies and policy discussions and interviewed developers about their expectations, communication, and decision-making regarding changes. We found that developers employ a wide variety of practices that shift or delay the costs of change within an ecosystem. Expectations about how to handle change differ substantially among the three ecosystems and influence cost-benefit tradeoffs among those who develop packages used by others (who we will call upstream developers), the developer-users of such packages (who we will call

downstream developers), and end-users. We argue that these differences arise from different values in each community and are reinforced through peer pressure, policies, and tooling. For example, long-term stability is a central value of the Eclipse community, achieved by their "prime directive" practice of never permitting breaking changes. This practice imposes costs on upstream developers, who may accept substantial opportunity costs and technical debt to avoid breaking client code. In contrast, the Node.js/npm community values ease and simplicity for upstream developers and has a technical infrastructure in which breaking changes are accepted, but signaled clearly through version numbering.

Our second study builds on and expands the scope of the first, investigating the prevalence of practices, and attitudes toward the ecosystems values from Study 1, in a larger set of 18 ecosystems. We combine several methods to accomplish this, including data mining of software repositories to identify those practices that leave visible traces, document analysis to identify policy-level practices that are stated explicitly, and a large-scale survey to ask developers about many other practices as well as the importance of various values within the ecosystem. In Study 2, we find that practices and values are indeed often cohesive within an ecosystem, but diverse across different ecosystems. We also find that even when ecosystems share similar values, they often achieve it in different ways, or sometimes fail to achieve it at all, promoting practices that are never widely adopted or do not work well. Together, our results provide a map of the distribution of values and practices across these ecosystems and allow us to examine the relationships between values and practices. Beyond these findings, we make our full anonymized results available to the research community, in hopes they will be useful in future studies, for example, by providing a basis for selecting cases with particular combinations of practices and values.

This work builds on and extends our previously published conference paper [6], including much of the material in Section 4. The data is available as an archived dataset [7] as well as an interactive web page.[1]

Our contributions include a description of breaking change-related values and practices in three ecosystems, a taxonomy of values and of practices, and a mapping of those values and practices across 18 ecosystems derived from a survey, data mining, and policy analysis.

2 CONCEPTS AND DEFINITIONS

Software ecosystems. For this study, we define *software ecosystems* as *communities built around shared programming languages, shared platforms, or shared dependency management tools, allowing developers to create packages that import and build on each others' functionality.* In line with definitions of Lungu [50] and Jansen and Cusumano [43], we focus on "collection[s] of software projects which are developed and which co-evolve together in the same environment" [50, p. 27], which have interdependent but independently developed packages, which generally share a technology platform or a set of standards [43]. Such ecosystems typically center on some means to package, version, and often host software artifacts, and to manage dependencies among them [1, 47, 51, 61, 74].

Note that the term "software ecosystem" is overloaded and used with different definitions in different lines of research [52], including ones that focus on commercial platforms that can be enhanced with third-party contributions [40, 56, 81, 83]. We focus especially on open-source communities developing interdependent libraries (e.g., *Maven*, *npm*, *CPAN*), rather than more centralized platforms where usually independent extensions provide a single application but do not build on each other (e.g., Photoshop plugins, Android apps); we also exclude ecosystems that repackage

[1] http://breakingapis.org.

software projects and their dependencies for deployment (e.g., Debian packages, homebrew), as they are often managed by independent volunteers rather than the original software developers.

Breaking changes. There are many relevant software development concerns when maintaining interdependent artifacts as a community. We focus on the coordination issue of deciding whether and how to perform *breaking changes* and how downstream developers respond.

In this article, we define a *breaking change* as any change in a package that would cause a fault in a dependent package if it were to blindly adopt that change. We thus include not only cases where a change in API would cause a downstream package to fail to compile, but also cases where program behavior would change, leading to incorrect results or unacceptable performance. We examine *breaking-change related practices* quite broadly, including not only reactions to actual breaking changes, but practices meant to signal, mitigate, or prevent breaking changes.

Maintaining dependencies and updating one's own code to react to breaking changes is a significant cost driver when using otherwise free open-source dependencies. Breaking changes are common in practice [3, 5, 6, 14, 22, 29, 39, 44, 48, 53, 54, 66–68, 89, 90]. For example, Decan et al. [22] found that 5% of package updates in CRAN were backward incompatible, causing 41% of the errors in released dependent packages. Xavier et al. [90] report that 28% of releases of frequently used Java libraries break backward compatibility, with the rate of breaking changes increasing over time. Information hiding [63], centralized change control [29, 73], and change impact analysis [8, 84] can all guide decision making, but cannot entirely prevent the need for breaking changes in practice, given the large-scale, open, and distributed nature of software ecosystems [6, 59, 62, 76, 90].

Package managers structure the problem and make dependencies and versions explicit [3, 47, 51], and practices like *semantic versioning* assign semantics to version numbers (e.g., breaking vs. nonbreaking changes) [65, 67], but these only help to manage change, not prevent the problem or support decision making about when to perform breaking changes.

Values and practices. The "why" and "how" of managing breaking changes in software ecosystems are *values* and *practices*.

Shared *values*—judgments of what is important or preferred—can explain how developers make similar decisions. Values have been studied at societal scale in psychology [4], ethics [16], and related fields [12, 37] (e.g., how education influences personal value systems); however, values and their influence on practices have been studied mostly in narrow contexts in software engineering: Pham et al. studied testing culture [64] and Murphy-Hill et al. found that creativity and communication with non-engineers is valued more by game developers than by application developers, resulting in less testing and architecture practices in game development [58]. We use the concept of values to analyze common shared beliefs about what is important for an ecosystem, with a focus on change-related issues.

With *practices*, we refer broadly to activities that developers engage in, again primarily with a focus on managing change. Practices may include specific release strategies, deciding not to perform changes, mitigating the impact of changes through documenting migration paths or reaching out to developers, monitoring changes in dependencies, deciding whether and when to update dependencies, and many more [6].

In ecosystems, practices may be encouraged or mandated by *policies* (for example, *npm* and *Eclipse* mandate the use of semantic versioning in their documentation) and may be supported or even enforced by *tools* (for example, the *Eclipse* community's *API Tools* detect even subtle breaking changes and *CRAN* runs automated checks to enforce coding standards and resolve incompatibility issues) [6]. For simplicity, we use the term practice broadly, including policies and tools.

Governance in open source and software ecosystems covers community-wide decisions, e.g., how to integrate third-party contributions [11], which model for decision making is generally appropriate [45, 60], how open an ecosystem should be [85], and how people in different roles should be allowed to participate [86]. While some governance research discusses the need for both evolvability and stability of an organization [83], research focuses on general market mechanisms or process documentation and conformance [41, 45] not on technical steps a software engineer might take.

3 METHODS

3.1 Research Design

As stated in the introduction, our goal in this research is to create a high-level map of values and practices relating to breaking change across many software ecosystems

We approached this question with an exploratory sequential mixed-methods design [15], beginning a qualitative preliminary case study to first understand *how* the community deals with or prevents breaking changes, and *why* they deal with them in this way. This first study takes a constructivist view, focusing on how the problem of breaking changes look from the perspective of participants, and asking why they approach this collaboration problem the way they do. We use this to inform a second, primarily quantitative study. The second study is not intended specifically to *confirm* that the findings generalize (although we do a confirmatory check in Section 5.1), but rather a broad look to see *where* it generalizes, and if there is any pattern to the combinations of values and practices we see in the larger landscape outside the three case study ecosystems. Study 2 casts a broad net at the cost of depth when asking high-level questions about many communities; however, we recognize and call for research about particular practices, values, or ecosystems that should be followed up in more depth, bringing more resources to bear for more focused questions. Study 2 shows that there is not a simple relationship between practices and values—we found that communities often act on the same value in different ways.

3.2 Study 1: Interview Case Study

For our first look at ecosystem practices, we performed a multiple case study, interviewing 28 developers in the three ecosystems. Case studies are appropriate for investigating "how" and "why" questions about current phenomena [92]. We selected three contrasting cases to aim for *theoretical replication* [92], a means to investigate the proposition that phenomena will differ across contrasting cases for predictable reasons.

Eclipse and Node.js/npm served as cases that contrast sharply in their approach to change: Eclipse has interfaces that have not changed for over a decade, while Node.js/npm is a relatively new and fast-moving platform. We expected that Eclipse's policies and tools might impose costs on developers in a way that encouraged them to act consistently with the ecosystem's values of stability. The R/CRAN ecosystem serves as a useful third *theoretical replication*, since its policy favors compatibility among the *latest* versions of packages over Eclipse's long-term compatibility with *past* versions. In addition, CRAN acts as a gatekeeper for a centralized repository in contrast to npm's intentionally low hurdles for contributions.

We began by mining lists of packages and their dependency relationships from these three ecosystems. We assembled a database of packages, their dependency relationships, and version change histories from the npm repository (metadata from which was retrieved from https://registry.npmjs.org/ in json format), CRAN repositories (scraping metadata from web pages starting from http://cran.r-project.org/web/packages/available_packages_by_name.html), and git repositories of Eclipse (https://git.eclipse.org/c/).

Table 1. Interviewees. R2 and N4 Were Pairs of Close Collaborators, Identified as R2a, R2b, N4a, and N4b

Code	Case	Field	Occupation
E1	Eclipse	Programming tools/HCI	University
E2	Eclipse	Soft. Eng./CS Education	University
E3	Eclipse	Soft. Eng./Research	University
E4	Eclipse	CS Education	University
E5	Eclipse	Software engineering	Retired
E6	Eclipse	Software engineering	Industry
E7	Eclipse	Eclipse infrastructure	Industry
E8	Eclipse	Software engineering	Industry
E9	Eclipse	Software engineering	Industry
R1	CRAN	Soil science	Government
R2a,b	CRAN	Statistics	University
R3	CRAN	Medical imaging	University
R4	CRAN	Genetics	University
R5	CRAN	Soil science	University
R6	CRAN	Web apps	Industry
R7	CRAN	Data analysis	Industry
R8	CRAN	R infrastructure	Industry
R9	CRAN	R infrastructure	Industry
R10	CRAN	R infrastructure	University
N1	NPM	Telephony	Industry
N2	NPM	Tools for API dev.	Industry
N3	NPM	Web framework	Startup
N4a,b	NPM	Web framework	Startup
N5	NPM	Cognitive Science	University
N6	NPM	Database, Node infrastr.	Startup
N7	NPM	Database, Node infrastr.	Industry

All owned packages with both upstream and downstream dependencies.

We pursued two complementary recruitment strategies for our interviews. To find package maintainers who would have recent, relevant insight about managing dependencies from both sides of the dependency relationship, we used our mined repository datasets to identify packages that had at least two downstream dependencies and two upstream dependencies, and that both the focal package and at least one of the upstream dependencies had had a version update in the year before the interview (2015).[2]

We emailed a random sample of these packages' owners choosing at random from the package list mentioned above in small batches, handwriting emails to the authors using emails and details supplied in the npm and CRAN repositories, or the Eclipse commit logs, and set up interviews with people who responded. We also interviewed three developers that we or our colleagues knew personally. In all, we contacted 92 people and conducted 26 interviews. Our interviews focused on their personal practices and experiences managing upstream and downstream dependencies.

After 20 interviews, we were hearing similar ideas from each new interviewee but we recognized the need for deeper experience with the ecosystem-wide origins and impacts of the ecosystem's

[2]The code implementing this filtering is available at https://github.com/cbogart/depalyze/blob/1d867cc92d7a5f18274358ae02574915026a30d5/depalyze/versionhistory.py#L354.

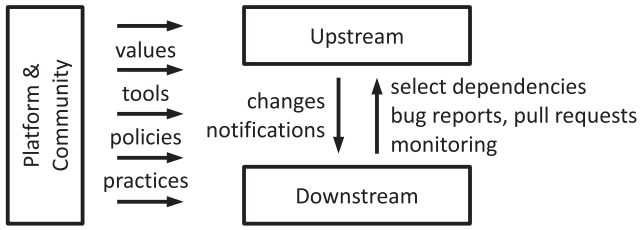


Fig. 1. Conceptual overview: *upstream* vs. *downstream* and influence of platform and community.

policies, so we decided to additionally interview individuals with some role (current or historical) in the development of the ecosystem's tools or policies. As these individuals are fewer and there are more demands on their time, we only attempted to find a few key people in each ecosystem; thus, we recruited 8 additional developers; asking a few of the same questions but also adding questions about the ecosystem's history, policy, and values. All 28 interviewees were active software developers with multiple years of experience, but their background ranged from university research to startup companies; Table 1 gives an overview.

We conducted semistructured phone interviews that lasted 30–60 minutes. We generally followed an interview script shown in Appendix A, but tailored our questions toward the interviewees' personal experiences. With the interviewees' consent, we recorded all interviews.

In keeping with our constructivist approach to the first study, we analyzed the interviews using Thematic Analysis [9]. We transcribed the recordings, then tentatively coded the transcripts looking for interesting themes, using Dedoose [23], then iteratively discussed, redefined, and recoded. Codes that emerged in the first round included labels such as "expectations towards change," "communication channels," "opportunity costs of backward compatibility," and "monitoring." We combined redundant codes, eliminated ones that did not recur or address our research questions, then grouped the remainder into seven high-level themes: "Change planning: reasons for changes," "change planning: costs to the changer," "Change planning: Technical means, practices," "Change planning: reasoning about cost tradeoffs," "Coping with change," "Communication," and "Ecosystem-wide policy and technology." Next, we gathered tagged quotes from each high-level category, and two researchers checked that they agreed with the low-level tags for each quote in the category, revising any disagreements through discussion.

Thematic analysis does not claim to find reproducible phenomena within the interviews; for example, we did not attempt to compute interrater reliability, since we make no claim that two researchers trained themselves to reliably identify exactly the same utterances from interviewees as examples of "expectations towards change," nor that we have exhaustively identified all instances of such an expectation among our interviewees. As such, we do not apply statistics to our qualitative results or attach much importance to counts; the purpose of the interviews and our thematic analysis is to discover the broad categories of attitudes and strategies towards change that interviewees experienced, with illustrative examples of typical practices and motivations that constitute those strategies.

To complement our interviews, we explored policies, public discussions, meeting minutes, tools in each ecosystem.

In our analysis, we distinguish between decisions made in the roles upstream and downstream developer, as depicted in Figure 1.

Validity check. To validate our findings from the case study, we adapted Dagenais and Robillard's methodology [18] to check fit and applicability as defined by Corbin and Strauss [13, p. 305]. We presented interviewees with both a summary and a full draft of Sections 4.2–4.3, along with

questions prompting them to look for correctness and areas of agreement or disagreement (i.e., fit), and any insights gained from reading about experiences of other developers and platforms (i.e., applicability).

Six of our interviewees responded with comments on the results; all six indicated general agreement (e.g., *R5: "It brings a structure and coherence to issues that I was loosely aware of, but that are too rarely the centre of focus in my everyday work."*); corrections included small factual errors, (e.g., the number of CRAN packages had increased since the initial writeup, and is now over 14,000); and suggestions of ways to sharpen our analysis (e.g., R7 noted that CRAN's policy to contact downstream developers does not apply to the majority of users outside CRAN). We incorporated their feedback when it was consistent with a recheck of our data and added clarifications otherwise.

3.3 Study2

We then conducted a systematic mapping of values and practices in a broad sample of ecosystems, primarily making use of a survey. Because of the large number and diversity of practices (Tables 4, 5, and 6), we could not measure them all with one methodology. We asked about a large subset of them in the survey (e.g., doing research about dependencies before using them; bottom section of Table 6). We also analyzed documentation and policies to identify practices that are enacted ecosystem-wide by organizations or tools (e.g., Ecosystem-wide synchronized release; Table 4); finally, we mined Github repositories and the libraries.io package metadata dataset for practices that leave visible traces (e.g., "Continue critical updates to older versions"; Table 5). Out of the 55 practices we identify, there are 19 that we do not attempt to measure in Study 2 (e.g., socially connected developers following each other on Twitter, going to conferences; top section of Table 6).

First, we describe the survey methods, then in subsequent subsections describe the policy analysis (Section 3.3.5) and data mining (Section 3.3.6) methods.

3.3.1 Ecosystems. We solicited survey participants from ecosystems with a *dependency network structure*, in which packages can depend on other packages and a standardized infrastructure helps with sharing and compatibility. We started with a list of software repositories from Wikipedia's "Software Repository" page and added additional ecosystems with an active community that we could find.

We excluded ecosystems with a flat structure where packages depend only on a single shared platform (e.g., Android) and ecosystems obviously too small to hope to get at least a few dozen responses. We also excluded ecosystems if they were different enough that it was not possible to write clear questions that would apply across ecosystems. This excluded, for example, operating-system-level package managers such as apt, rpm, and brew, and scientific workflow engines.

We conducted the survey with 31 ecosystems. For our analysis, we somewhat arbitrarily set the minimum number of participants for each ecosystem at 15, feeling this would give us a reasonable claim to some breadth in the responses. This led us to exclude 13 ecosystems: C++/Boost, Bower, Perl 6, Smalltalk, Tex/CTAN, Julia, Clojure/clojars, Meteor, Wordpress, SwiftPM, PHP's PEAR, Racket, and Dart/pub, leaving us with 18 ecosystems for our analysis, shown in Table 2. All but 2 had more than 40 complete responses.

3.3.2 Survey Goals and Recruitment. The survey consisted of 108 questions: seven long free text questions (marked as optional opportunities for clarification), three short text questions (ecosystem, package name, and gender), and the rest multiple-choice scales. After an informed consent screen, participants first were asked to choose an ecosystem in which they had published or used a package (they could choose from a list, or type in another; we grouped rare answers as "other" for analysis).

Table 2. Survey Statistics by Ecosystem

Ecosystem	C/S	Role	Age	Mean	St.dev.
Atom (plugins)	41/74			28	9
CocoaPods	47/109			31	9
Eclipse (plugins)	43/79			36	9
Erlang,Elixir/Hex	44/74			32	7
Go	45/142			35	10
Haskell (Cabal/Hackage)	46/89			32	10
Haskell (Stack/Stackage)	17/40			30	6
Lua/Luarocks	16/27			33	13
Maven	42/84			38	9
Node.js/NPM	86/235			29	8
NuGet	52/97			34	8
other	55/376			37	13
Perl/CPAN	44/70			42	10
PHP/Packagist	54/166			31	8
Python/PyPi	94/244			33	9
R/Bioconductor	48/71			35	7
R/CRAN	53/97			37	13
Ruby/Rubygems	57/122			33	8
Rust/Cargo	48/125			29	10
		u s c l + f	2 3 4 5		

Statistics about survey takers: C/S = Surveys Completed/Started; Survey taker's age: 2 = 18 - 25, 3=26-35, 4=36-45, 5=46+; Survey taker's role: u=user, s=submitter, c=committer, l=lead of a package in ecosystem, +=lead of core package, f=founder. Mean and standard deviation of age are estimated assuming each survey taker's age was in the center of the surveyed range.

3.3.3 Recruitment. We invested in significant outreach activities to recruit participants for the survey. First, we created a web page and Twitter account to describe the state of current research in this area, in a form easily accessible to practitioners.[3] We encouraged readers of the web page to take the survey to contribute additional knowledge about values in ecosystems. Second, we attended community events, including *npm.camp 2016*, to talk to developers and community leaders from multiple ecosystems about our research; as a result, several prominent community members tweeted about our web page and survey, resulting in surges of responses (*CRAN* and *npm* particularly). Third, we promoted our web page and the survey in ecosystem-specific forums and mailing lists to *"developers who write <ecosystem> packages,"* hoping that our web page would spark interest in the topic. We also posted on Twitter with hashtags appropriate for different ecosystems. Finally, for 21 ecosystems in which our outreach activity did not yield sufficient answers, we solicited individuals directly by email. We sent 8,137 emails to package authors. We sampled these from authors of packages culled from libraries.io for targeted ecosystems.

Participants and their demographics. We succeeded in recruiting 2,321 participants to partially or fully complete the survey between August and November of 2016. Of this number, 932 completed the survey; however, we put value questions near the beginning, so there are 1,466 answers to those questions. Statistical analysis of answers to early questions did not reveal any systematic differences between people who completed the survey and those who did not (mean difference between answers to 65 Likert-scale questions between respondents who completed the survey and

[3] https://breakingapis.org.

those who did not, was 0.13 scale points (out of 4 or 5, depending on the question). The maximum difference was .83 scale points; but the maximum difference among questions where more than one "incomplete" respondent answered was .54 Likert-scale points). Since the partial responses were similar to full responses, we include data for the incomplete responses.

To correct for careless responses in which people appeared to be answering many questions without careful consideration, we excluded as "careless" those sections of a person's response in which they rated all items exactly the same. We performed this test on eight sections of the survey, and the number of excluded blocks ranged from 11 (for a set of upstream practices) to 76 (for a set of downstream practices). When people were excluded from one block, their responses to other questions did not appear to be outliers (mean difference between answers to 65 Likert-scale questions between respondents excluded from some other block, and respondents who were not, was 0.15 scale points (out of 4 or 5, depending on the question). The maximum difference was .50, for the question "How important do you think the following values are to the <ecosystem> community: stability"). Because the answers were similar for all questions, we did not exclude entire people if they were apparently careless in any of the eight blocks.

Table 2 shows participation by ecosystem. Participants averaged 8.8 years of development experience, 7.2 years in open source, and 4.6 in the ecosystem they answered about. Slightly more than half (59%) had college degrees in CS. The most frequently claimed role in the ecosystem was package lead developer (59%); Others ranged from the 8.5% who claimed a role in the founding or core team of the ecosystem, to 11% who only drew on ecosystem packages for their own projects. The average age was 33, with 152 18–24-year-olds, and 6 over 65. Of those who gave their gender, 95.9% identified themselves as male, 3.2% as female, and 0.8% gave another gender. These demographic proportions are quite similar to a contemporaneous Github community survey [31].

3.3.4 Survey Design. Our goal in the survey was to investigate the prevalence of values and practices across as many ecosystems as was feasible. We asked a larger number of questions than is typical for a survey of this sort. Long surveys often have reduced completion rates, however, we mitigated this by keeping the questions diverse and hopefully interesting to the participants, and by putting the questions we were most interested in up front. As a result, we got a reasonably high completion rate (40%) and partial completion rate (62% for value questions at the beginning) considering the length of the survey, resulting in an encouragingly rich and deep dataset. In this article, we focus on describing the values and practices responses, but additional data is available in the accompanying data release [7].

Values. To explore as complete a list as possible of values relevant to managing change, we began with values derived from our interviews in Study 1. We then searched each of the web pages of all our candidate ecosystems for clues of other potential values. For example, "*fun*" is mentioned as an explicit value in the Ruby community; in an interview Ruby founder Matsumoto said, *"That was my primary goal in designing Ruby. I want to have fun in programming myself" [82].* Note that some values initially seem not directly related to breaking change, but we included them if we thought they could indirectly influence breaking change practices. For example, we expected that perhaps if some practices are more efficient, but less rewarding to carry out, then a "fun"-valuing ecosystem might avoid them.

We assembled a list of 11 values with the following descriptions:

- *Stability:* Backward compatibility, allowing seamless updates ("do not break existing clients").
- *Innovation:* Innovation through fast and potentially disruptive changes.

- *Replicability:* Long-term archival of current and historic versions with guaranteed integrity, such that exact behavior of code can be replicated.
- *Compatibility:* Protecting downstream developers and end-users from struggling to find a compatible set of versions of different packages.
- *Rapid Access:* Getting package changes through to end-users quickly after their release ("no delays").
- *Quality:* Providing packages of very high quality (e.g., secure and correct).
- *Commerce:* Helping professionals build commercial software.
- *Community:* Collaboration and communication among developers.
- *Openness* and Fairness: Ensuring that everyone in the community has a say in decision-making and the community's direction.
- *Curation:* Selecting a set of consistent, compatible packages that cover users' needs.
- *Fun* and personal growth: Providing a good experience for package developers and users.

In the survey, we asked participants about the *perceived values* of the community—*"How important do you think the following values are to the <ecosystem> community?"* We used a seven-point rating scale, adapted from Schwartz's value study [71]: *"extremely important," "very important," "important," "somewhat important," "not important," "community opposes this value,"* and *"I don't know."* The first five options were separated visually from the last two to make clear that only the former were designed to approximate regular intervals (as recommended by Dillman et al. [27]).

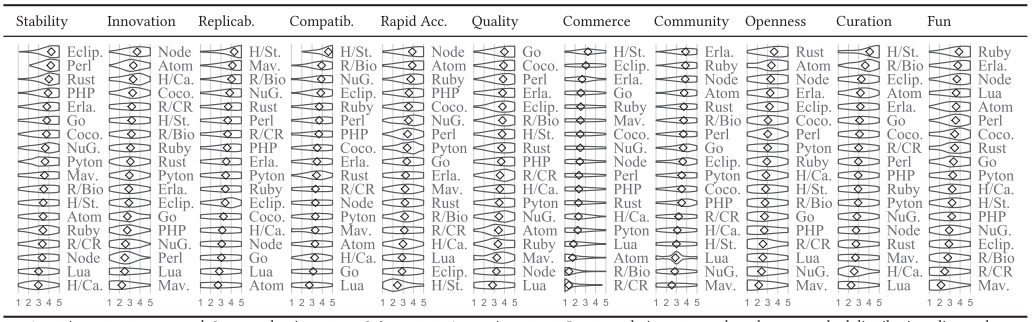
In addition, we asked participants a similar value question on the same scale about their *own values* with respect to a single package they worked on in the ecosystem. To encourage participants to think about concrete work that they are doing, we asked for the name of a specific package that they worked on and used that package in the question: *"How important are each of these values in development of <package> to you personally?"*

Recognizing that despite taking values from multiple sources, we may not have captured all values relevant to managing change, we asked survey participants in an open-ended question about other values important to their ecosystem. Their answers are summarized in Section 5.2.

Practices. The *practices* part of the survey asked about many software-engineering practices, many of which we mention throughout our analysis (Tables 4, 5, and 6); the full list and exact phrasing of our questions can be found in Appendix B. Surveyed practices encompassed the participant's personal practices and experiences with respect to documentation, support, timing, and version numbering for releases, selecting packages on which to depend, and monitoring dependencies for changes. These were asked, as appropriate, either on an agreement Likert scale as above or on a frequency scale from *"never"* to *"several times a day."* A subset of 15 questions relating to communication with developers of downstream packages were skipped for participants who indicated that they did not maintain a package used by others. To limit the length of the survey, we focused primarily on questions that cannot be answered or are difficult to answer by mining software repositories or reading explicit policy documents (see "M" and "P" labels in Tables 4, 5, and 6) in the Study 2 Methods column.

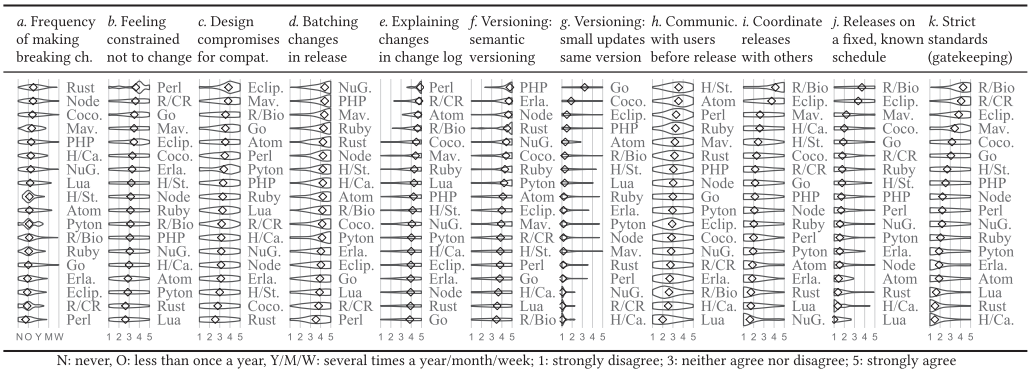
Survey analysis. 483 participants (21%) gave an answer to at least one of the seven optional free-response questions; 11 people gave answers to all seven. We used a grounded approach to analyze answers to the question about other values: one researcher performed open coding to identify a set of candidate codes, then two researchers iteratively combined and revised these to achieve a consensus set of codes and to apply them to the responses.

Layout of Figures. Figures 2, 3, and 4 were drawn by eliminating skipped or "don't know" values, merging "Not important" with "opposed to this value" answers, and drawing a violin plot, with a



1: not important or opposed, 2: somewhat important, 3: important, 4: very important, 5: extremely important; plots show smoothed distribution; diamond indicates mean

Fig. 2. Perceived community values; showing distribution of raw ratings, sorted by mean value, to emphasize range of answers.



N: never, O: less than once a year, Y/M/W: several times a year/month/week; 1: strongly disagree; 3: neither agree nor disagree; 5: strongly agree

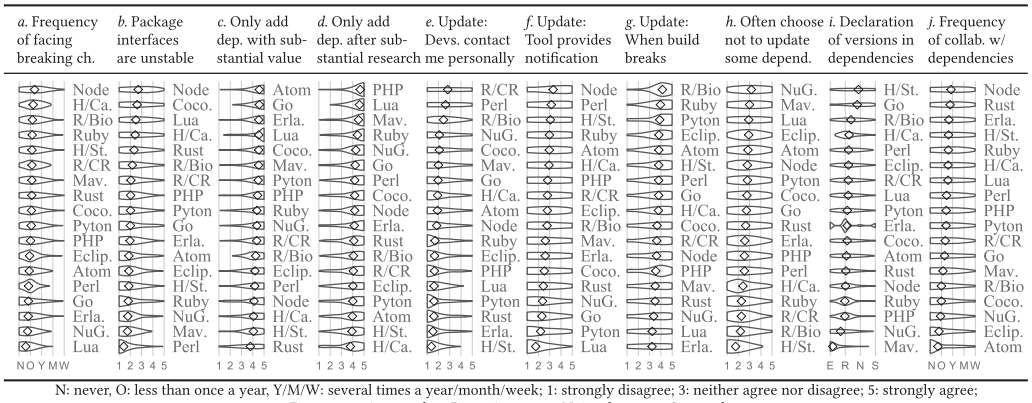
Fig. 3. Practices of package maintainers and frequency of performing breaking changes.

diamond symbol at the mean position. The violin bodies are smoothed, so the image portrays the mean and a rough distribution.

For Table 10, we wanted to derive a ranking of the importance of the values in each ecosystem and provide an indication of the consensus around the ranking. The method we adopted calculates highest ranked values for each ecosystem by identifying, for each person in the ecosystem, their highest rating of any of the 11 values, then incrementing a count for *all* values that person assigned that highest rating to. This has the effect of counting the number of people who ranked each value as the highest while accounting for ties. The table lists the values with the three highest counts, and the consensus numbers are as described in the caption.

3.3.5 Policy Analysis Method. We examined each ecosystem's online presence and summarized their sanctioned practices. Practices of the ecosystems were derived from documentation pages within each language's and repository's websites, specifically seeking out documentation about how to define a package and submit it to the repository, as these documents typically communicate policies to authors in a clear, actionable way. The columns of the table were defined as follows:

- **Dependencies outside repository.** Standard tools in all but two ecosystems (Stackage and LuaRocks) allow developers to additionally specify packages that are not part of the standard repository, for example by a reference to a GitHub repository or an alternate specialized site. We checked the documentation for each package manager's syntax about how



N: never, O: less than once a year, Y/M/W: several times a year/month/week; 1: strongly disagree; 3: neither agree nor disagree; 5: strongly agree; E: exact version number; R: version range; N: just by name; S: snapshot

Fig. 4. Practices of package users and frequency of facing breaking changes.

to declare dependencies, to see if there was a way to specify a URL for a package not formally in the repository. We marked these as having the feature if it could be specified directly as a URL; as "alternate repo" if this could be accomplished only through an alternate repository, or a custom server that mimics the repository's API.

- **Central Repository.** This captures whether the ecosystem supplies packages in a central repository or simply provides an index to author-hosted download sites.
- **Access to dependency versions.** This denotes whether ecosystem documentation recommends (through examples in the documentation page) for packages to refer to dependencies by version number, or to simply assume the latest version of a dependency is desired (R/CRAN and Go).[4] In two cases (Stackage and Bioconductor), a set of mutually compatible versions is provided to be used together as a set.
- **Gatekeeping Standards.** Ecosystem repositories vary in the amount of vetting of the packages they include. We determined this by looking at the submission requirements for packages. An open circle in the table means that no more than cursory metadata such as name of the package and list of dependencies are required; a closed circle means that platform tools or volunteers perform some deeper investigation of the package: vetting of the submitter, automated or manual tests (of the package or of other packages that depend on it), or virus checks. Two were marked as "staged releases," because submissions are tested collectively along with a cohort of packages being released simultaneously.
- **Synced Ecosystem.** This simply denotes whether ecosystem packages (or some important subset) are released all at once on a regular, synchronized schedule.

3.3.6 Data Mining. We mined data from two sources to capture data about the prevalence of seven additional practices.

First, the list of packages to query was derived from the libraries.io (libraries.io/data) cross-ecosystem package index. Libraries.io lists versions, their release dates, dependencies with their version constraints, and their source repositories. It was only available for a subset of our 18 ecosystems (Atom, R/CRAN, Perl/CPAN, Ruby/Rubygems, Rust/Cargo, Python/Pypi, NuGet, Maven, PHP/Packagist, Node.js/NPM, Erlang,Elixir/Hex). Partial information was available for CocoaPods

[4]Recommendations have evolved since 2016 for Go: see https://blog.gopheracademy.com/advent-2016/saga-go-dependency-management/.

Table 3. Ecosystem Statistics

Ecosystem	Founded	Num. Pkgs	Avg. deps	>3 deps	>0 deps
Atom (plugins)	2014	4,424	1.2	10.0%	38.2%
CocoaPods	2001	14,493	0.4	1.7%	21,1%
Eclipse (plugins)	2001	14,954	6.4	55.7%	100%
Erlang,Elixir/Hex	2013	1,304	1.0	5.3%	50.5%
Go	2013	76,632	10.6	57.1%	88.3 %
Haskell (Cabal/Hackage)	2003	8,593	6.4	57.9%	91.6%
Haskell (Stack/Stackage)	2012	1,337	8.3	65.0%	93.9%
Lua/Luarocks	2007	966	0.8	5.7%	34.7%
Maven	2002	114,404	2.1	20.6%	41.8%
Node.js/NPM	2010	229,202	5.6	49.8%	81.2%
NuGet	2010	66,486	1.6	11.4%	58.3%
Perl/CPAN	1995	31,641	7.6	56.5%	79.6%
Python/PyPi	2002	65,622	0.2	2.0%	8.1%
PHP/Packagist	2012	63,860	3.1	28.1%	82.7%
R/Bioconductor	2001	1,104	4.9	48.9%	74.2%
R/CRAN	1997	7,922	2.9	27.9%	86.7%
Rust/Cargo	2014	3,727	2.1	20.1%	71.5%

Package dependency and founding year data for ecosystems. # Pkgs = number of packages in the repository we checked as of January 2016; Avg. deps = average number of dependencies sampled packages had; >3 deps = percentage of packages having more than three dependencies. >0 deps = percentage having any dependencies.

and Hackage, but not dependencies. Dependency counts for Bioconductor, Hackage, Stackage, Lua, Eclipse, and CocoaPods were scraped from their respective repository websites. We did not find Go dependencies listed centrally in any repository, so we extracted this information from World of Code [57], a massive mirror of GitHub, GitLab, Bitbucket, and other open source software repositories, indexed and searchable in ways that make it more convenient for data mining than GitHub's APIs allow. One data product World of Code provides is dependencies of packages, parsed from source code files; we used this to count Go dependencies. Table 3 shows that packages in the ecosystems are interdependent, but in widely differing degrees.

Beyond package counts and dependencies, further information about these packages was queried about packages in all ecosystems from World of Code [57].

- **Dependency Version Constraints.** We ran pattern-matching on the dependency constraints of all packages in libraries.io, for packages released during 2016 and flagged for each package whether it used a particular type of constraint on any one or more of its dependencies at any time during the year. Note that percentages add up to over 100%, since a package may use more than one kind of dependency constraint.
  - —*Exact*: Dependency version is constrained by a fully specified version number, such as 1.3.2.
  - —*Min only*: Version constraints such as >1.3.2, or use of conventions like caret (^) in npm that has the same effect (e.g., ^1.3 is the same as >= 1.3.0).
  - —*Range*: Constraints with a minimum and maximum version, like >1.3.2,<2.0; or use of conventions like tilde (~) in npm that has the same effect (e.g., ~1.3.2 means >=1.3.2,<2.0).

— *Unconstrained*: The dependency name is specified with no version constraints; either the constraint is blank or some symbol like "*" is used. [5]

For a more fine-grained analysis of version constraints across many ecosystems, see Dietrich et al. [26].

- **Lock files.** Using World of Code [57], we examined files committed during 2016 in each of the ecosystem's packages, looking for references to a lock file, which specifies exact versions of all dependencies, direct and transitive (i.e., dependencies of dependencies). These differ by ecosystem and vary in how canonical their use is. The filenames we used in this search are shown in Table 11 in Appendix D. Including a lock file in an end-user distribution of a program makes it more likely the program will run correctly, since it preserves the exact versions of dependencies that the program was tested on. However, developers including many dependencies in their own projects may prefer not to specify the exact versions of all their transitive dependencies, since they may be in conflict with each other, and they have the means and opportunity to resolve any conflicts themselves (then perhaps locking in a consistent set of dependencies when producing a release for their own users) [78].
- **Maintaining old versions.** Making bug fixes to outdated versions of code, or even backporting new features, can be helpful for users who cannot update to the cutting-edge versions for some reason. We define prior-version maintenance operationally as simply any release whose version number is smaller than expected and hence out of sequence: For example, if a sequence of releases was "2.0.1," "2.0.2," "1.5.3," "2.0.3," then we identify "1.5.3" as a likely bugfix or backported feature introduced in 2.0.1 or 2.0.2, introduced as a courtesy to those users currently using 1.5.2 who choose not to upgrade to the 2.0 series. Specifically, this measure captures the percentage of packages in each ecosystem whose version number ever decreased in 2016, per data from Libraries.io.
- **Cloning.** We measured the percentage of packages in each repository whose projects borrowed a file in 2016 from another package. We did this by building a list of SHA hashes of files (blobs) associated with each commit in each project in the ecosystem through World of Code [57], and looking for overlaps. We count a project as having cloned a file, if a commit incorporates a blob over 1 kb in 2016 that was previously seen in some other package in the ecosystem. We only considered blobs derived from other packages in the ecosystem's repository, not ones derived from projects in the broader realm of open source. We chose to count these within-repository clones specifically, since the developer could have tried to use the ecosystem's dependency management system to incorporate the desired code by reference, but chose not to. Previous research has also mapped cloning behaviors [33, 49].

3.4 Threats to Validity

We chose our methods carefully to answer our research questions, and the survey in particular differs from a more typical statistically focused survey technique. We therefore describe the threats to the validity of the study before presenting the results, so readers can have these in mind as they read our findings.

As described, Study 1 used case selection criteria [92] appropriate for contrasting cases, but they may not be typical of all ecosystems, and so one needs to be careful when generalizing beyond the three cases. Our results may be affected by a selection bias, in that developers who did not want to be interviewed may have had different experiences. Finally, the differences we found among cases

[5]Note that this weighs most heavily the state of packages for which more versions were released or that had more dependencies.

may be confounded with the reasons we selected them, such as their popularity or the availability of data about them.

As for Study 2, as is typical of surveys in our field, our survey sample is not truly random; there may be selection bias relating to who we were able to reach via the venues we chose. We tried to mitigate this by recruiting from forums, Twitter, and direct e-mail. The survey was also quite long (and was advertised as such up front). People with less patience for long surveys, or less interest in questions of breaking changes, values, and practices, may have self-selected out. This could be significant if people with impatience for long surveys also have different software-engineering practices and beliefs.

Another possible concern is that respondents may apply different standards in their ratings. For example, if the expectation of stability is extremely high in a particular ecosystem, then participants may rate the perceived importance of stability lower, because they are applying a very stringent standard for how focused everyone should be on stability. A similar focus on stability in a different ecosystem might lead to participants in that ecosystem to rate the importance of stability higher. We tried to mitigate this by requiring at least 15 participants for each ecosystem, which should give some breadth of experience behind the responses.

While we tried to avoid using terminology that differed among ecosystems, we were not always successful. For example, the word "snapshot" means different things in different ecosystems' practices, which caused some confusion. Even the term "breaking change" may be interpreted differently; for example, they might define it more narrowly as a change that simply would cause downstream compilation to fail, while we intended it to also include changes that would cause wrong behavior in downstream software.

Respondents may also have given answers to a few questions influenced by social desirability. For example, they may have felt obliged to say that "quality" is extremely important because that is the "right" answer, or that people follow certain practices because they are what they know to be expected. Our mitigation approach was ensuring confidentiality of responses and avoiding, to the extent possible, questions with clear desirable and undesirable responses.

We had difficulty recruiting sufficient participants from smaller ecosystems, such as Perl 6 or Clojure; small ecosystems may have different characteristics than large ones. We do have two small ecosystems, Stackage and Lua, and they are outliers in some ways. So, further exploration of small ecosystems, for example with interviews or analysis of artifacts, should be a priority for future work.

4 STUDY 1: QUALITATIVE MULTIPLE-CASE STUDY

In Study 1, we investigated the decision-making involved in making breaking changes, and practices they adopt to ease the burden:

RQ1.1: How do developers make decisions about whether and when to perform breaking changes and how do they mitigate or delay costs for other developers?

We also wanted to see how developers responded to breaking changes that affected them:

RQ1.2: How do developers react to and manage change in their dependencies?

Finally, we wanted to know whether developers perceived tensions between platform policies and their intended effects:

RQ1.3: Did platform policies or tools ever have unintended consequences?

4.1 Case Overview

To understand the identified different practices and policies, it is important to understand the purpose and history of each ecosystem. In the following, we provide a brief description of all

three ecosystems and their values, informed by both public documentation and our interviews. Platform-level features or practices relevant to breaking change are identified in Table 4.

4.1.1 Eclipse. The Eclipse foundation publishes more than 250 open source projects. Its flagship project is the *Eclipse IDE*, created in 2001. The IDE is built from the ground up around a plugin architecture, which can be used as a general purpose GUI platform and in which plugins can depend on and extend other plugins. Projects can apply to join the Eclipse foundation through an incubation process in which their project and practices come under the Eclipse management umbrella. It is also common practice to develop both commercial and open-source packages separately from the foundation, and publish them in a common format on a third-party server. In addition, the "Eclipse marketplace" is a popular registry, listing over 1,600 external Eclipse packages that can be installed from third-party servers through a GUI dialog.

The Eclipse foundation coordinates a "simultaneous release" of the Eclipse IDE once a year and (as of 2016) three "update releases" for new features in between. Many external developers align with those dates as well.

The Eclipse foundation is backed by corporate members, such as IBM, SAP, and Oracle. Its policies are biased toward backward compatibility; packages (e.g., commercial business solutions) developed 10 years ago will often still work in a current Eclipse revision without modification.

A core value of the Eclipse community is backward compatibility. This value is evident in many policies, such as *"API Prime Directive: When evolving the Component API from release to release, do not break existing Clients"* [25]. Although not entirely uncontroversial (as we will explain), this value was confirmed by many interviewees.

4.1.2 R/CRAN. The ***Comprehensive R Archive Network (CRAN)*** has managed and distributed packages written in the R language since 1997. R is an interpreted language designed for statistics. The R language itself is updated approximately every six months, but new development snapshots are available daily. R has multiple repositories with different policies and expectations, including Bioconductor and R-Forge; we focus on CRAN, the largest one. CRAN formally exists under the umbrella of the *R Foundation*, but sets its own policies.

CRAN contains over 8,000 packages. Of these, 29 are either required or "recommended," and are bundled in binary installs. About 2,200 more are cataloged as useful for 33 different specializations such as finance and medical imaging. Distributing R software as a CRAN package gives it high visibility, since installation from CRAN is automated in the command-line version of R and the popular IDE *RStudio* [69].

R and CRAN are used by many developers without a formal computer-science or programming background. CRAN pursues *snapshot consistency* in which the newest version of every package should be compatible with the newest version of every other package in the repository. Older versions are "archived": available in the repository, but harder to install. When a new package version is submitted to CRAN, it is evaluated by the CRAN team's partly automated process. The package must pass its own tests and must not break the tests of any downstream packages in CRAN that depend on it without *first* alerting those package's authors so they can make corresponding fixes. Package owners need to react to changes in the platform or in upstream packages within a few weeks, otherwise their package may be archived.

A core value of the R/CRAN community is to make it easy for end-users to install the most up-to-date packages. Although not explicitly represented in policy documents, this value was apparent from many interviews; for example, R10 said, *"CRAN primarily has the academic users in mind, who want timely access to current research."*

Table 4. Platform and Community-level Practice Choices: Who: (P)latform, (U)pstream, (D)ownstream, (3) Third party; Study 2 Method: (P)olicy Analysis, (S)urvey, (M)ining

Who	Study 2 Method	Practice
P	P	Existence of centralized repository or directory of packages
P	P	Mechanism for referring to dependencies distributed outside official repositories (e.g., via github directly)
P	P	Make historical versions of package easy or difficult to rely on
P	P	Mechanism to remove or reassign unmaintained packages (e.g., maintainers do not respond to emails)
P	S	Releasing changes on a fixed, advertised schedule per package
P	S,P	Ecosystem-wide synchronized release
P	P	Repository personnel check standards of submitted code before making available on the repository
P		Allow multiple versions/only one version of a package to be loaded at the same time
P/U		"Stability attributes" (in Rust) saying which API points will not change
P		Use nightly unstable builds to get exciting new features (at cost to compatibility for downstream users)
P		Disallow wildcard dependencies
P		Test compiler changes against all published software using it to prevent breaking things
P		Constrained rules about version numbering (e.g., cargo disallowing wildcards)
3	P	Third-party curation of sets of useful packages or compatible versions
P		Dynamic language feature to help backward compatibility (optional parameters in R)
P		Centralized testing infrastructure for all packages
P		Vulnerability tracking (e.g., Node security platform)
U	S	Private arrangement among package authors to release at the same time

For ecosystem-by-ecosystem breakdown of policies, see Section 5.

4.1.3 Node.js/npm. *Node.js* is a runtime environment for server-side JavaScript applications released initially in 2009, and *npm* is its default package manager. *npm* provides tools for managing packages of JavaScript code and an online registry for those packages and their revisions. The *npm* repository contains over 250,000 packages with rapid growth rates.

The Node.js/npm platform has the somewhat unusual characteristic that multiple revisions of a package can coexist within the same project. That is, a user can use two packages that each require a different revision of a third package. In that case, *npm* will install both revisions in distinct places and each package will use a different implementation.

A core value of the Node.js/npm community is to make it easy and fast for developers to publish and use packages. In addition, the community is open to rapid change. Ease for developers was one of the principles motivating the designer of *npm* [75]. Therefore, *npm* explicitly does not act as a gatekeeper; it does not have review or testing requirements; in fact, the *npm* repository contains a large number of test or stub packages. The focus on convenience for developers (instead of end-users) was apparent in our interviews.

4.2 Study 1 Results: Planning Changes (RQ1.1)

We first discuss managing change from the perspective of a developer planning to perform changes that may affect downstream users. While we observed similar forces and concerns regarding

change across all three ecosystems, we observed differences in how the community values affect the ways package maintainers mitigate or delay costs for downstream users.

4.2.1 Breaking Changes: Reasons and Opportunity Costs. Although breaking changes to APIs are costly to downstream users in terms of interruptions and rework, our interviewees gave many reasons why they had to perform such changes; there are corresponding *opportunity costs* that arise when deciding *not* to perform the change, such as the cost of maintaining obsolete code, working around known bugs, or postponing desirable new features.

Obvious and expected reasons for breaking changes included *requirements and context changes* and *rippling effects* from upstream changes. Beyond that, we found surprisingly frequent mentions of stylistic and performance reasons, as well as difficult bug fixes.

Technical debt. Surprisingly, 12 interviewees (E3, E9, R1, R3, R4, R5, R6, R7, R8, N1, N7) mentioned concerns about technical debt, rather than bugs, new features, or rippling upstream changes, as the trigger for breaking changes. By *technical debt*, we refer to code that is functionally sufficient but has outstanding stylistic issues developers want to fix, such as poorly chosen object models or method names, lack of extensibility or maintainability, or little-used or long-deprecated methods.

We conjecture that the reason interviewees brought up these kinds of changes so often in discussion was because they had thought about them in depth. Technical debt often arises from the tension between tools and practices that encourage developers to preserve backward compatibility (e.g., Eclipse's "prime directive"), versus general pressure for evolution and improvement. Developers often postpone breaking changes until the technical debt becomes intolerable; for example, E3 mentioned as the reason for planning to finally remove some deprecated code: *"What we did there was to provide old methods as deprecated. But that gets quite messy. At one point almost half of the methods were deprecated."* E9 similarly told us about an upcoming long-postponed major version change: *"since we don't do it often, probably once every five years, [...] let's take advantage of that opportunity to do some of the things that would be good that we couldn't do before."*

Old interfaces can come to seem old fashioned and unattractive in a swiftly changing community. Three interviewees said they made breaking changes for syntactic reasons: to harmonize syntax (R1) or improve *"weird"* or *"bad"* names (R3, R4) in their interfaces. N7 talked about adopting a new JavaScript programming paradigm that was far more attractive: *N7: "You can't just stay on that old stuff for forever, it's just not going to work. And so we drastically rewrote the internals at the transport to be a stream, because that's sort of, essentially what it is, right? Like, it's a little stream that takes logs and sends them places."* However, four interviewees (E1, E5, E6, R6) talked about the consequences when *not* being able to make such changes, i.e., having to preserve old interfaces over long periods, caused opportunity costs, since it hindered attracting new developers, lured by cutting-edge things. E6, for example, told us that: *"If you have hip things, then you get people who create new APIs on top of that in order to [for example] create the next graphical editing framework or to build more efficient text editors. These things don't happen on the Eclipse platform anymore."*

Efficiency. Four interviewees (E6, R1, R4, N1) reported cases in which efficiency improvements required breaking changes. For example, N1's package offered an API for requesting paged data that the server could not provide efficiently; they deprecated and eventually removed that function rather than spending money on hardware.

Bugs. Bug fixes were another reason for breaking changes (E4, E7, R7, R9). Bug fixes can break downstream packages if those packages depend on the *actual* (broken) behavior instead of the *intended* behavior. A lack of well-defined contracts in most implementations makes assigning blame and responsibilities difficult in practice. As E5 told us, *"If someone likes the broken*

semantics, then they're not going to like the fixed semantics." Thus, even fixing an obvious mistake in code under the control of a single person can require significant coordination among many people.

Throughout our interviews, we heard many examples of how bug fixes effectively broke downstream packages, and the difficulty of knowing in advance which fixes would cause such problems. For example, R7 told us about reimplementing a standard string processing function and finding that it broke the code of some downstream users that depended on bugs that his tests had not caught. R9 commented on the opportunity cost of not fixing a bug in deference to downstream users' workarounds for it: *"If the [downstream package] is implemented on the workaround for your bug, and then your fix actually breaks the workaround, then you sort of have to have a fallback … [pause] It gets nasty."*

4.2.2 Dividing and Delaying Change Costs. Our previous discussion already hinted that there is flexibility regarding who bears the costs of a breaking change. For instance, a package's developer can decide between making a breaking change, pushing costs for rework to maintainers of downstream packages; or *not* making the change, accepting opportunity costs such as technical debt. Even when deciding to make the change, the developer faces strategic choices about whether to invest more effort when making the change to reduce the *interruption and rework costs* for downstream users as well as to *affect timing* of when those costs are paid (Table 5). For example, by documenting how to upgrade, the developer invests more effort to reduce effort for downstream maintainers. Different developers and different communities have different attitudes toward *who* should pay the costs of a change and *when*, as we will show.

Awareness of Costs to Downstream Users. Almost all (24 out of 28) of our interviewees stated that, when possible, they avoid breaking changes that would affect downstream users. Reasons included looking out for their users' best interests and knowing that costs to affected users would come back to them, as users ask for help adapting to the change, ask for the change to be reverted, or seek alternative packages. Two interviewees (E1 and R4) specifically mentioned concern for downstream users' scientific research (R4: *"We're improving the method, but results might change, so that's also worrying—it makes it hard to do reproducible research"*).

Interviewees' concern for impacts on users was tied to the size and visibility of the user base and the perceived importance and appropriateness of their usage. Nine interviewees across all ecosystems (E4, E5, E6, R1, R4, R6, R7, R9, N7) were aware of their users and were concerned specifically about the number of users affected and the quantity of complaints that a change would imply, e.g., *R9: "I wanted to rename it to something that more specifically describes that this is actually a new V8 context, but, you know, I can't because so many packages are already importing the new context function." N1: "we happen to know that paging is not the feature that was […] often used from Node module customers"* Another npm developer said, *N7: "… that was strictly a breaking change for [feature], and so we really didn't want to break all the community [feature]. Like, we didn't want all 700 of these to give out 'the code you're using, you have to upgrade … Good luck, bro."'* An R/CRAN developer said, *R7: "I'm very cautious about making changes to it, and then when I make changes I often regret it. Even for a small change on a package used by a lot of people, it improves 90% of people's lives, but makes 10% of people's lives worse, and 1% complain, which, with [package] can be a lot of people."* Three interviewees (E1, R4, R8) noted that their sensitivity toward avoiding breaking changes grew with experience and with a growing user base, as they learned from feedback received about earlier breaking changes.

Of course some developers also themselves work on such downstream packages. Four of our interviewees mentioned doing so (E5, N4, N7, R6) (see discussion in Section 4.3.1); these are presumably aware of the impact of the changes they make to their own other packages.

Only four developers were not particularly worried about breaking changes. Three (E6, N1, N5) had strong ties to their users and felt they could help them individually (N5: *"We try to avoid breaking their code—but it's easy to update their code"*). Interviewee N6 expressed an "out of sight, out of mind" attitude: *"Unfortunately, if someone suffers and then silently does not know how to reach me or contact me or something, yeah that's bad but that suffering person is sort of [the tree] in the woods that falls and doesn't make a sound."*

Finally, developers described tradeoffs in fixing mistakes that downstream users had come to depend on. E8 talked about being stuck with a poor design *"If you make a mistake in your API [. . .] sorry, you're stuck with it, so you have to kind of work around it."* R9 mentioned circumstances where users depended on buggy behavior, but the upstream code had to be fixed anyway: *"After upgrading the parser some people complained that their script was no longer working. But the problem was that their syntax was invalid to begin with. It's obviously their fault."*

Techniques to Mitigate or Delay Costs. Despite a strong general preference for avoiding breaking changes, there are many cases where the opportunity costs of not making a change are too high. Our interviewees identified several different strategies for how they, as package maintainers, routinely invest effort to reduce or delay the impact from their changes for downstream users.

Maintaining old interfaces. Across all ecosystems, preserving the old interface alongside a new one is a very common approach to mitigate an immediate impact of a change on downstream users. While specifics depend on the language and tools, common strategies to avoid breaking downstream implementations include documenting methods as *deprecated* and providing default implementations for new extension points or parameters. In these strategies, the package developer invests additional effort *now* to preserve backward compatibility, accepting technical debt in the form of extra code to maintain for some time, in exchange for preventing an *immediate* downstream impact of the change. The developer may at some later time clean up the code, affecting downstream users that have not updated in the meantime [68].

Similarly, many interviewees (E2, E3, E5–E8, R1, R6–R9, N1, N7) told us about various techniques to perform changes without breaking binary compatibility. They prevent rework costs for existing users by accepting more complicated implementations and harder maintenance in the changed package, while possibly also creating costs for new downstream users who have to deal with more complicated mechanisms.

Parallel Releases Seven developers (E5, E6, R1, R2, R4, R7, R8) reported strategies to maintain multiple parallel releases, such that downstream developers can incorporate minor nonbreaking changes (e.g., bug fixes) without having to adopt major revisions. Node.js/npm's caret operator allows package authors to support parallel releases with different version numbers: An author can publish an update 1.0.1 to their version 1.0.0, even after 2.0.0 has been released; users who wish to stay with the 1.* series but still receive updates may refer to version `^1` or `^1.x` to receive anything less than 2.0.0.[6] It is a common practice to provide security patches[7] including for older releases.[8] In contrast, CRAN only supports sequential version numbering,[9] causing some developers to fork their own packages (e.g., `reshape2` was introduced as backward incompatible revision to `reshape`). However, R8 told us this is discouraged by CRAN: *R8: "Because <package>2, it's the second version of <package>, at what point can you just freeze an API and leave it there, and*

[6] https://docs.npmjs.com/misc/semver.
[7] Current npm security alerts are listed at https://www.npmjs.com/advisories.
[8] e.g., https://www.npmjs.com/advisories/1482.
[9] According to https://cran.r-project.org/web/packages/policies.html, "Updates to previously-published packages must have an increased version."

jump n+1 version and just continue with that? I think there's some lingo in [CRAN's instructions for package authors] that they'd rather not have that." In each case, the fact that they are adding code to multiple versions suggests that developers are investing significant additional effort to reduce the (immediate) impact on downstream users. For example, N1 told us that they were conservative about making major new versions, since their package *"has changed major version numbers a lot over last few years, many things backported to earlier versions; irritating to do major revisions every couple of months."*

A variant of this strategy is to maintain separate interfaces for different user groups with different stability commitments within the same package (see the façade pattern in Reference [30]). For example, interviewee E5 provided in parallel both a detailed and frequently changing API for expert users and a simpler and stable API that insulated less sophisticated users from most changes. Similarly, interviewee R1 has split packages into smaller packages, with the intention that each user could depend only on parts relevant to them and would be exposed to less change. In both cases, the developer accepts the higher design and maintenance costs of multiple APIs for reduced impact on specific groups of users with distinct needs.

Release Planning. Individual developers and communities may take consideration of downstream users by planning *when* to release changes. R1 keeps versions of his package with a quickly changing API in a separate repository and batches multiple updates together in CRAN less frequently when he wants to release a version to a broader audience. While in R/CRAN and Node.js/npm packages are released by individuals whenever they want, the core packages of the Eclipse community coordinate around synchronized yearly releases[10] (a strategy also common in other package systems such as *Debian*[11] and *Bioconductor*).[12] Delaying releases may incur coordination overhead and opportunity costs in slowing down development for the changer, but reduces the frequency (though not necessarily the severity) with which downstream users are exposed to changes and gives downstream users a planning horizon.

Communication with users. Finally, developers communicate in various ways with users to reduce the impact of a breaking change. Seven interviewees (E6, R4, R7, R8, R9, N6, N7) made early announcements to create awareness and receive feedback. R7 explained that *"two weeks or a month before the actual release, I do sort of a pre-release announcement on Twitter [and] tell people to use the README."* He told us during the validation phase that he has since written a script to email all downstream maintainers before a release.

Another reason for communicating with downstream users was to help them deal with the aftermath of change. In the simplest case, a developer could invest effort in documenting how to upgrade. Nine interviewees (E7, R2, R3, R7–R9, N1, N4, N5) mentioned being aware of their users personally, and could reach out to them individually; for example, N1 contacted users who were still using an old API, to help them migrate, and N5 had most users present on-site and could therefore help them migrate their code. E7 went so far as to create individual patches for all downstream packages within the Eclipse core to get them to adopt a new interface and move away from an old deprecated one. In all these cases, package maintainers invest effort to reduce costs for downstream users.

4.2.3 The Influence of Community Values. The previously discussed techniques are mechanisms that developers can use for tweaking who pays for the costs of a change and when. Individual developers often adopt patterns and, in fact, six interviewees (E1, R3, R4, R5, R8, N6) described gradual

[10] https://wiki.eclipse.org/Simultaneous_Release.
[11] https://www.debian.org/doc/manuals/debian-handbook/sect.release-lifecycle.ro.html.
[12] According to https://www.bioconductor.org/developers/package-submission/, "There are two releases each year, around April and October."

Table 5. Practices (Mostly Upstream) to Communicate and Mitigate Effects of Change

Who	Study 2 Method	Practice
U	S	Freeze APIs to protect downstream users from change
U		Release a major change as a new package name, rather than a new version
U		Mark API points as deprecated to warn of future removal
U		Remove deprecated API points eventually
U		Parallel releases to protect users who do not want to upgrade
U	S	Release changes in a batch rather than as they are made, to make less churn for users
U	S	Write new code as backward compatible, possibly at the cost of incurring technical debt
U	S	Proactively notify users about upcoming changes
U	S	Assist users who are having trouble upgrading to a new version with a breaking change
U	S	Write a migration guide to help users upgrade
U	S	Write a change log to document compatibility problems with prior releases
U	S	Use semantic versioning to signal the kinds of changes being made
U/P	S	Platform rules requiring package authors to negotiate compatibility before releasing (snapshot consistency)
U	M	Continue critical updates to older versions, to give users a way to avoid an expensive major upgrade
U/P		Ways to check that APIs have not changed, e.g., API tools, @since tags, documentation

adoption of more formal processes over time, as they learned their value through experience. At the same time, we could clearly observe that attitudes and practices differ significantly among the three ecosystems and are heavily influenced by ecosystem values, tools, and policies.

Eclipse. Developers are willing to accept high costs and opportunity costs to further Eclipse's value of backward compatibility, especially for core packages. The community has developed educational material explaining Java's binary compatibility and giving recommendations for backward compatible API design [24, 25]. With *API Tools*,[13] the community has developed sophisticated tool support to detect even subtle breaking changes and enforce change-related policies, such as adding `@since` tags to API documentation. Breaking changes in core packages are in fact very rare [38].

Even though they arguably make the platform harder to learn and maintain, Eclipse developers have identified and documented [25, part 3] workarounds for extending an interface while *maintaining old interfaces*, such as creating additional interfaces to avoid modifying existing ones (e.g., `IDetailPane2`, `IDetailPane3`, `IHandler2`) and runtime weaving. Deprecating interfaces and methods is common, but actually removing them is not[14]; for example, like many other methods, `org.eclipse.core.runtime.Plugin.startup()` as of this publication was still included despite being deprecated for over 15 years.[15] E6 noted that this backward compatibility prevents modernizing APIs, such as replacing arrays with collections.

[13] https://www.eclipse.org/pde/pde-api-tools/.

[14] e.g., a guide published by the Eclipse foundation about evolving APIs says that, "Obsolete API elements should be marked as deprecated and point new customers at the new API that replaces it, but need to continue working as advertised for a couple more releases until the expense of breakage is low enough that it can be deleted." [25].

[15] This method was deprecated in 2004: https://github.com/eclipse/eclipse.platform.runtime/commit/a46e757a1938edb0a7109dafef349c3a3ffc58ea and was still present in 2020: https://github.com/eclipse/eclipse.platform.runtime/blob/9aedff3f2141631a8bc5fa6d1abe005ea633f107/bundles/org.eclipse.core.runtime/src/org/eclipse/core/runtime/Plugin.java.

The Eclipse community invests significant effort into release planning, at the cost of some resulting friction, as reported by multiple interviewees. *E9: "Eclipse has a release process, and some projects have to release at the same time as the platform, some projects the day after, some projects the day after, [so] you're expected to be available a little bit before, so you can make sure that yours bills properly right? [. . .] So, that's kinda a complexity."* The required coordination is invested toward ensuring stability and smooth transitions at few plannable times for downstream users. An Eclipse release is a complex process with steps aimed at maintaining not only technical interoperability with prior versions, but also maintaining a consistent level of legal compatibility, usability standards, security, and so on.[16] This culture of conservative change contrasts with what, for example, an R developer told us: *R7: "On one hand I try to be careful, but on the other hand I don't want to inflict harm and be like paralyzed by the fact that anything I do might make someone's life worse. Sometimes you have be like go ahead and accept that things are going to break and it's not the end of the world."*

In Eclipse, maintenance releases for old major revisions are not common (Table 7); presumably because with backward compatibility users can simply be told to update to the latest release.

R/CRAN. As the R/CRAN community values making it easy for users to get a consistent and up-to-date installation, developers invest significant effort to achieve consistency.

There is no policy against CRAN packages making changes that affect the larger body of code *outside* of CRAN. However, when changes affect other CRAN packages, upstream developers are asked to bear the significant extra cost of reaching out to and coordinating with maintainers of affected packages[17] (termed "forward impact management" by De Souza and Redmiles [19]). Downstream maintainers then may also bear the cost of pressure to update their packages first before the upstream developer can make a breaking change, to ensure that all CRAN packages are consistent. CRAN's policy requires (and verifies) that developers maintain constant synchronization with each other, and 5 of our 10 interviewees (R2, R3, R7, R8, R9) specifically mentioned reaching out individually to known, downstream developers (in contrast to three Node.js interviewees (N1, N4, and N5) and one Eclipse interviewee (E7)). Synchronization is thus continuous, but more decentralized and localized than with Eclipse's simultaneous releases.

Among our interviewees, five developers of specialized R packages targeted small and close communities and knew their users personally. For example, R3 mentioned that "no one used" a feature, and when asked how they knew that, they replied that *"statisticians working on a lot of medical imaging [. . .] type of applications in R is a very small community. There's only so many people to know."* R3 said he got to know those users because of interactions about the dependency. Only a one of our Node and Eclipse interviewees (E6) mentioned personal connections with downstream users, but our sample is too small to be sure this is not just sampling bias.

Consistency is enforced by manual and automated checks on each package update.[18] The change management process is collaborative but also demanding of a maintainers time; R7 said the timeline to adapt to an upstream change *"might be a relatively short timeline of two weeks or a month. And that's difficult for me to deal with because I try to sort of focus one project for a couple weeks at a time, just so I can remain productive."* Node developers, in contrast, can ignore changes until they feel like updating (*N5: "Why don't we upgrade more often? It's more work than you'd hope."*), while Eclipse developers rarely need to worry about change (e.g., *E1: "When a new version comes*

[16] https://wiki.eclipse.org/Development_Resources/HOWTO/Release_Reviews.

[17] https://cran.r-project.org/web/packages/policies.html#Submission "If an update will change the package's API and hence affect packages depending on it, it is expected that you will contact the maintainers of affected packages and suggest changes, and give them time (at least 2 weeks, ideally more) to prepare updates before submitting your updated package."

[18] https://cran.r-project.org/web/packages/policies.html#Submission.

out every year in July or whenever, I'd go ahead and test if my plugin works correctly in that new version; if it does, I don't care much about that. [...New features] were mostly irrelevant. I didn't care that much about that.")

The platform is not conducive to multiple parallel releases—on CRAN a package revision must have a higher version number than the one it supersedes, so an old major version cannot be updated; policies also discourage forking a project and submitting it with a separate name.[19] There is no central release planning, perhaps because it is perceived to slow down access to cutting-edge research.

Overall, we observed much more communication and coordination with downstream users about individual changes than in Eclipse, but also more flexibility with regard to performing breaking changes.

Node.js/npm. The Node.js/npm community values ease for upstream developers and the possibility to move fast [75]. It is much less demanding for a developer to make a breaking change. Six of the Node.js interviewees talked about the importance of signaling change through *semantic versioning*.

This sharply contrasts with the R developers we asked about this: two R interviewees spoke out against semantic versioning; for example, *R7: "I'm familiar with the semantic versioning stuff. It's just I don't find that useful personally, because most R users aren't familiar with that and I think [convention] is a little bit on the ridiculous side. [...] For most R users I don't think version numbers send a terribly strong signal, and they are likely to not know what version they are using currently anyway."*

Semantic versioning in Node allows developers to make breaking changes as long as they clearly indicate their intentions. Because the technical platform allows downstream developers to still easily use the old version without fearing version inconsistencies, breaking changes do not as easily cause rippling effects or immediate costs for downstream users. While they still avoid breaking changes and employ various strategies to maintain old interfaces, in our interviews, Node.js/npm developers were generally willing to perform breaking changes in the name of progress and in fighting technical debt, including experimenting with APIs until they are right. For example, N6 told us that if a downstream user was concerned about a breaking change: *"I could tell this person, well look if you have this problem at least for now your workaround is very simple. Change your dependency to be this exact dependency so instead of saying we depend on package* `foo`*version* `*`*. Change it to just exactly that version [...], and you will still be using the old one that you know and love. And that will postpone your problem until the day that you need some new thing that's come out which is no longer backported into the old version. [...] So knowing that, I do kind of feel kind of confident enough to just say yeah we're gonna bump the major version, we're gonna announce or whatever that takes, but I don't really myself feel too much desire to kind of read for the backward compatible people."*

As mitigation strategy, maintenance releases for old versions are common, made easy by the platform and associated tools. Analyzing the npm repository, we found that 24 of the 100 most "starred" packages did this at least once; this was more common than in Eclipse or R/CRAN (Table 7).

Summary of RQ1.1 results: Developers are motivated to change code for many reasons, such as requirements and context changes, bugs and new features, rippling effects from upstream changes, and technical debt from postponed changes. There are also opportunity costs from

[19] https://cran.r-project.org/web/packages/policies.html#Submission.

forgoing or postponing changes. Opposing this motivation is their awareness of costs to downstream users of such changes, especially when their userbase is large and visible to them; in most cases developers want to avoid imposing those costs on users. Their choice is not binary, however; there are ways of softening the impacts of change, such as maintaining old interfaces, making parallel releases, and making and communicating plans about upcoming changes. Developers weigh these choices differently depending on the ecosystem's values: Eclipse core package developers are discouraged heavily against change, and thus opt for techniques to allow strictly backward-compatible additions. R/CRAN developers are not officially discouraged from making changes, but they are aware that the ecosystems rules (no parallel releases, onus on downstream users to update) are burdensome for downstream users, so they emphasize communication and collaboration in their updates. Node.js/npm developers are encouraged to make changes, by mechanisms that signal downstream users about changes, yet insulating them from the requirement to adopt the changes; as a result upstream developers are quite likely to opt for change, and to police each others' rigorous use of the signaling mechanisms for change (semantic versioning).

4.3 Study 1 Results: Coping with Upstream Change (RQ1.2)

Just as upstream developers have some flexibility in planning changes that may affect downstream developers, downstream developers have flexibilities regarding *whether, when, and how to react to upstream change*, again influenced by values, policies, and technologies (Table 6). Having to monitor and react to upstream change can be a significant burden on developers (e.g., mismatch between schedules has been shown to be a barrier to collaboration [42]). The *urgency* of reacting to change can depend significantly on the development context and platform mechanisms.

When discussing how frequently they react to upstream change, our interviewees described a spectrum ranging from never updating (E3) to closely monitoring all changes in upstream packages (N1, N2, R9). Two interviewees mentioned explicitly ignoring certain upstream changes (N3, N7); others upgraded dependencies only at the time of their own releases (N3, N5) or during deliberate house-cleaning sweeps (N7, E2). Even when the platform does not require updates, developers often prefer to update their dependencies to incorporate new fixes and features (E3, N2) or to avoid accumulating technical debt (R6, N5). But they avoid updating when updates require too much effort (e.g., by causing complicated conflicts; N5, E3) or cause too much disruption downstream (N7).

4.3.1 Monitoring Change. When developers have to or want to react in a timely fashion to upstream changes, they need to monitor the upstream projects in some way. The platform itself, e.g., Node.js, R core, and the CRAN infrastructure, is often an additional source of changes that developers need to keep up with. In our interviews, we discovered many different strategies for monitoring, including technical and social strategies. Their strategies varied along with the urgency of their needs, from active monitoring of upstream activity, to general social awareness of upstream activities, to a purely reactive stance where developers wait for some kind of notifications.

Active monitoring. Only four interviewees (E5, R9, N1, N4) reported actively monitoring upstream changes, in the sense of maintaining personal awareness of upstream changes, by regularly looking at activity going on in their upstream dependencies. R9, N1, and N2 said they used GitHub's notification feed with some regularity (N2 only for changes to the Node.js platform, not to upstream packages). N4 kept up by following Twitter feeds, blogs, and attending conferences. R7 indicated that raw notification feeds, in their current form, are a significant burden with a low

signal to noise ratio, saying that *"The quantity of notifications I get on GitHub [on my own project] already is to the point of overwhelming. So I don't even mostly read them unless I'm actually working on the project at that moment."* He later told us that after our interview he tried scaling back to watching just the three to five projects he is actively working on. Only one interviewee (R9) did not feel overwhelmed, saying that occasional skimming of GitHub feeds was useful way to get an overview of activity.

Upstream participation. In seven cases, developers mentioned monitoring upstream changes not as outsiders following a stream of data, but as active participants in those projects, collaborating to influence them toward their own needs (E5, N4, N7, R6) or providing direct contributions to those packages (E7, E9, R7). For example, in describing the challenge of getting upstream projects to prioritize changes that he needed, an Eclipse developer said, *"I touch everything that I care about, because it's really hard to convince other people to do things that I need to do. I find it much easier to just learn all the projects and when I need something, to do it myself."* This aligns with de Souza and Redmiles' observation of exchange of personnel as a common strategy for cooperation among dependent projects[19]. Such developers wear hats in both projects: They maintain active awareness of the upstream project, as downstream developers, and as upstream developers, their downstream work informs their understanding of the upstream project's requirements.

Others like E5 actively compiled and tested their project with development versions of upstream dependencies, emphasizing the importance of giving timely reactions: *"if you report it within a week there's a better chance the developer might remember what they did [...] which provides a good chance that they can revert their change before they hit their milestone."*

Social awareness. Many interviewees tried to maintain a broad awareness of change through various social means. The most frequently mentioned mechanism, especially in the Node.js community, was *Twitter* (E9, R7–R9, N2, N3, N4a, N4b, N6, N7). For example, N4a commented, *"the people who write the actual software are fairly well connected on Twitter, [...] like water cooler type of thing. So we tend to know what's going on elsewhere."* In each ecosystem, interviewees (E5, R9, N4, N6) mentioned the importance of face-to-face interactions at conferences for awareness about important changes in the ecosystem. Other mentioned social mechanisms to learn about change were personal networks (R6, R8), blogs (E1, R4, R7, R8, N4, N7), and curated mailing lists (N1).

Reactive monitoring. Although our research questions led us to probe interviewees about the aforementioned active and social monitoring practices, a *reactive* strategy is also possible for dependencies. That is, rather than maintain some awareness and understanding of plans and activity in an upstream project, for example, by watching a Github feed and keeping track of why they follow each project and which changes might be relevant to them, a developer may instead ignore upstream projects' activity until they are given actionable evidence that their own project needs to adapt in some way. The developer waits to hear about problems from others (in advance, or after things had broken): Upstream developers contacting them about breaking changes, failing tests after dependency updates, or platform maintainers warning of changes that would affect them. There are tools that enable this reactive stance, that generate targeted notifications on certain kinds of changes. The specific tools differ among the platforms and support different practices or policies. Policies and common practices (e.g., testing practices) in the platform strongly in turn affect the reliability of a reactive strategy and corresponding tools.

Four developers (R3, E5, N2, and N7) mentioned the use of continuous integration to detect compile-time issues caused by breaking changes in upstream packages early. The tools *gemnasium* [32] and *greenkeeper* [35] allowed Node.js/npm developers to get notifications about new

Table 6. Practices (Mostly Downstream) to Monitor Change and Manage or Avoid Its Effects

Who	Study 2 Method	Practice
Awareness and coordination		
D	S	Reactively track what upstream packages are doing (when it breaks; when you're notified somehow)
D	S	Proactively track (maintain awareness via github notifications, mailing lists, etc.)
D	S	Submit feature requests and bug reports to upstream package authors
D	S	Participate in decision-making about upstream package's future
D	S	Tool-based notifications about upstream changes (e.g., Greenkeeper)
D		Regularly test against unreleased development versions of dependency to give timely feedback
P		Socially connected group of developers following each other on Twitter, going to conferences, etc.
P		Political work among core people to get buy in on making a breaking change
Protection against each potential change		
D	S	Do not update dependencies; just leave them at old versions known to work
D		Upgrade dependencies all at once only when making a new release
D	S	Dependency hell: manual manipulation of dependency version constraints to get a set of dependencies to be mutually compatible
U	S	Violate semantic versioning for trivial changes to prevent rippling updates that version change would require
D	M	Lock file: fix versions of all upstream packages (incl. transitive dependencies) with release
D		Report wrong semantic versioning as a bug
D	M,S	Specify an exact version number of a specific dependency
D	M,S	Specify a range of legal version numbers of dependencies (e.g., allow minor but not major upgrades)
D	M,S	Specify only a dependency's name and do not constrain what version of it is to be used
Protection against dependencies themselves		
D	S	Do significant research about each dependency weighing whether to adopt it
D	S	Wrap the dependency in an abstraction layer to decrease risk of change
D	S	Avoid use of dependencies, roll your own
D	S,M	Clone the dependency's code and maintain the new code yourself
D	M,S	Copy dependency code into your own repository ("vendoring") to get exact version needed

releases of upstream packages. Gemnasium alerted developers of package releases that fix known vulnerabilities, whereas greenkeeper submitted pull requests to automate a continuous integration run against the new release. In either case, developers could react to notifications by email or pull requests.

CRAN's requirement that upstream developers notify their downstream dependents when a change is coming appears to encourage downstream developers across the ecosystem to take a reactive stance (in contrast to Eclipse and Node.js/npm, where individual downstream developers need to employ optional monitoring tools). R7 defended the practice of waiting to be told about breaking changes as a principled attention-preserving choice, consistent with ecosystem norms; while R2 was apologetic about being reactive: *"I guess I'll sound crass about this and say it. For things*

like that I would wait to hear from CRAN when something broke. Because I don't think I can keep up with all of it." CRAN enforces this policy with manual and automated checking on each package update, running the package's tests and the test of all downstream packages in the repository, as well as some static checks. The CRAN team may then warn an affected downstream developer of an upcoming change by email.

4.3.2 Reducing the Exposure to Change. Many developers have developed strategies to reduce their exposure to change from upstream modules and, thus, reduce their monitoring and rework efforts. The degree to which developers adopt such mitigation strategies again depends on the technology, policies, and values, as we will discuss.

Limiting dependencies. Most of the CRAN and Eclipse interviewees that we asked (11 interviewees: R1, R2, R3, R4, R6, R7, E1, E2, E4, E5, E9) felt that it was better to have fewer dependencies. Reasons for limiting dependencies included limiting one's exposure to upstream changes and not burdening one's users with a lot of modules to install and potential version conflicts ("dependency hell"). Interviewee E5 represents a common view: *"I only depend on things that are really worthwhile. Because basically everything that you depend on is going to give you pain every so often. And that's inevitable."* Apart from removing no longer needed dependencies (tooling provided in Eclipse), six developers described more aggressive actions to avoid dependencies, including copying (R4) or recreating (R1, R6, R7, N6) the functionality of another package. N6 had to fork and recreate an upstream dependency as a temporary measure because of a licensing issue, but he did not feel dependencies were a burden generally.

In contrast, due to Node.js/npm's ability to use old versions and Eclipse's stability, three developers (E3, N1, N5) specifically said that they did not see dependencies as a burden.

Selecting appropriate dependencies. When limiting themselves to appropriate dependencies, interviewees mentioned a variety of different signals they looked for; these fell into five categories:

- *Trust of developers:* Seven interviewees (E4, R1, R5, R6, R7, N4, N6) mentioned basing decisions on personal trust of package maintainers. Criteria included being a large organization (E4), having a reputation for high quality code (R6, N6), and being consistent with maintenance (R6). One interviewee (R7) deliberately sent bug reports to a package to test whether the developer would be responsive before depending on it.
- *Activity level:* Five interviewees (E4, N6, N2, R1, R6) considered the activity level of the community of developers; for example, distinguishing a "real" ongoing project from an abandoned research prototype. Both high and low activity levels can be a positive indicator depending on the state of the project, as stated by N2: *" Ones with activity are mostly better maintained; they have lots of people contributing, like express. It's likely the community will have eyes on the ball, consider backward compatibility, ramifications [. . .]. Ones with little activity are small projects that don't change often, so change isn't an issue either."*
- *Size and identity of user base:* Four developers mentioned the size of the user base was using signals such as daily download counts (E2, N3, N5), whether projects of trusted developers use it (N6), or, as E2 said, *"Whether I'll actually jump on it or not is about how I perceive other software projects are using it."* N5 told us, *"We look to see how many people are using it: number of downloads per day. If it's low, that's a clue that it's sketchy, but not a perfect heuristic."*
- *Project history:* Four interviewees said they assumed that past stable behavior of a package would predict future stability (R1, R4, R6, E2). Signals included their own experience with the package (N4, E5), its status as part of the platform's core set of packages (E4), or its

visible version history, such as lack of recent updates and a version number above 1.0 (E3, N1, N4).

- *Project artifacts:* Finally, developers mentioned signals from project artifacts, including coding style (R1, R6), documentation (R1), good maintenance (N6), perceived ease of adoption (R1), code size (E2, N4, N7), and conflicts with other dependencies (N5).

Encapsulating change. Interestingly, there was almost no mention of traditional encapsulation strategies to isolate the impact of changes to upstream modules, contrary to our expectations and typical software-engineering teaching [63, 73, 88]. Only N6 mentioned developing an abstraction layer between his package and an upstream dependency, implemented because of an anticipated change. Questions about encapsulation were not in our interview protocol, so we did not ask about it specifically, but one possible explanation is that since upstream package already generally try to avoid gratuitous API changes, the ones that are necessary would require changes to an encapsulating class's API, obviating the point of the encapsulation.

4.3.3 Platform Values and Developer Values. Because policies, tools, and practices support different values in each ecosystem, they impose different costs on developers depending on whether their attitude towards some particular dependency aligned or conflicted with the community's broader values. In some situations developers will treat a dependency *as a fixed resource* to draw functionality from (also termed *API as contract* [20]), but in other situations, they treat the interface *as open to negotiation and change* (also *API as communication mechanism* [20]).

Eclipse's value on backward compatibility and predictable release planning is convenient for developers and corporate stakeholders who wish to rely on the released core platform code as a a fixed resource. Stability ensures that most developers relying on the platform packages do not need to monitor upstream changes, reacting at most to the yearly releases. Signals about whether to trust an upstream package are primarily social in the sense they can trust the packages that are part of the core, supported by corporations known to be invested in the stability of the platform.

According to E6, developers working within more volatile parts of the Eclipse ecosystem, such as using code outside the stable core, or in-development features of the core, have a greater need for monitoring and may be exposed to more change, sometimes encountering friction associated with that. E6 told us that *"there is a very different understanding of how important compatibility is and what it means, if you start from the platform, and then to the outer circles of Eclipse."* E5 talked about recompiling upstream code often to report bugs to them within a week. Thus, although Eclipse deeply values stability, there is necessarily a sphere of activity with active collaboration and change where that value is appropriately set aside.

CRAN's emphasis on consistency and timely access to research seems to encourage the *API as communication* rather than the *API as contract* [20] view of dependencies, in that its snapshot consistency approach forces maintainers to react to breaking upstream changes quickly (typically a few weeks [87]). This causes some apparent friction with researchers who might otherwise wish to publish their software and move on to other things. Many of the interviewees limited their dependencies, sometimes quite aggressively, by replicating code and reacting to notifications about change rather than actively following a community of upstream developers. However, an active and socially connected subset of developers (R7–R9) seemed to welcome collaboration. Although R7 advocated reacting to upstream changes rather than trying to anticipate them, R7, R8, and R9 emphasized Twitter and conferences to maintain an upstream awareness.

Node.js/npm's emphasis on convenience for developers has led to infrastructure that seems to decouple upstream and downstream developers from *having* to collaborate, since the downstream

can depend on old versions of the upstream for as long as they like. This should logically lead to *less urgency* to monitor upstream changes, except for patching security vulnerabilities. Developers do nonetheless often choose to take a collaborative approach to development, using tools such as continuous integration and *greenkeeper* [32] to force *themselves* to stay up to date despite the platform's permissiveness.

Summary of RQ1.2 results: Downstream developers are motivated to update their dependencies to take advantage of bug fixes and new features and avoid technical debt. However, such updates can be complex or risky, can disrupt downstream users, and may require some awareness of ongoing activity in an upstream project. Strategies to balance the costs and risks include different levels of awareness of upstream projects (from social or technical participation, to active or merely reactive monitoring), chunking the work by making all updating decisions at once periodically, or limiting the problem by carefully vetting dependencies to begin with. As with upstream change decisions, the ecosystem's context affects participants' choices. Eclipse's extreme interface stability allows downstream developers, at least outside the core, to trust it and ignore the possibility of change. CRAN's policy of global consistency among packages creates pressure for package maintainers to actively collaborate with their upstream counterparts; a core community seems to be spurred to active collaboration on Twitter and at conferences, while a peripheral community limits dependencies to avoid this necessity. Finally, NPM's tooling decouples downstream developers from immediate impact by upstream changes; developers who nonetheless wish to stay up to date adopt tools like greenkeeper to remind and encourage them to update.

4.4 RQ1.3 Unintended Consequences

Interviewees told us about instances where policies or their combinations led to unintended consequences.

Eclipse. One Eclipse developer said that the "political" nature of making changes can drive away developers and users. *"You have to be very patient and know who to talk with and whatnot; you really have to know how to play that game to get your patches accepted, and I think it's very intimidating for some new people to come on."* He explained that with many interdependent packages managed by different people each with a mandate not to change their interfaces, implementing a rippling change can require negotiations among people with conflicting interests.

Another consequence of Eclipse's stability, along with its use of semantic versioning, is that many packages have not changed their major version number in over 10 years. However, as E8 told us, strict semantic versioning is impractical to follow, so even for the few cases of breaking changes that are clearly documented in the release notes, such as removing deprecated functions, major versions are often not increased. Updating a major version number can ripple version updates to downstream packages, which can entail significant work for the many downstream projects that have hard-coded major version numbers for their dependencies.

Node.js/NPM. For Node.js/npm, in contrast, the rapid rate of changes and automatic integration of patches can raise concerns about reproducibility in commercial deployments. In many cases, the community then builds tools to work around some of the issues, such as providing tools that take a specific snapshot of an installation including all transitive package dependencies (e.g., "npm shrinkwrap" or R/CRAN's packrat). *"In npm, if you install today and tomorrow, you'll*

get 100s of dependencies, and something may have changed. So even if my version is the same, the servers could be running slightly different code, so customer facing code will differ and be hard to reproduce."

R/CRAN. CRAN has a similar issue regarding scientific, rather than deployment reproducibility: The community's goal of timely access to current research conflicts with many researchers' goal to ensure reproducibility of their studies [61].

In R/CRAN, the opposite dynamic from Node is evident in its versioning policy: The official policy on version numbers only requires that version numbers increase with each submission[20]; but a permissive form of semantic versioning is used and recommended by many developers [87, 91].

These conflicts and unintended consequences suggest that the design of ecosystem practices is not a solved problem.

Summary of RQ1.3 results: Unexpected community responses to policies included creative use of semantic versioning, innovative ways of promoting replicability, and stagnation.

5 STUDY 2: A SURVEY ON VALUES AND PRACTICES: PREVALENCE, CONSENSUS, AND RELATIONSHIPS

The research questions for Study 2 emerged in large part from the results of our first study. Study 2 endeavored to expand the scope beyond these three cases and to ask further questions raised by our results.

Study 1 revealed substantial differences in our three cases in the practices used to manage breaking changes and in the values these practices appeared to serve. This raises the question of how prevalent such differences are. Some values may be nearly universal, and some practices may be so fundamental, well-known, and effective that they are employed by nearly all ecosystems. However, different ecosystems make use of different technologies, have evolved different cultures, and serve different constituencies, suggesting that at least some values and practices may vary, perhaps dramatically, among ecosystems. Our questions for Study 2 were therefore:

RQ2.1: To what extent are values and practices for managing breaking changes shared *among* a diverse set of ecosystems?

Moreover, we have been making the assumption that ecosystems tend to have a shared view of values and practices across the ecosystem, i.e., that they are characteristics of ecosystems rather than individual projects or sub-ecosystem clusters of projects. It seems important to test this assumption, hence:

RQ2.2: To what extent do individual ecosystems exhibit consensus *within* the community about values and practices?

Finally, as we observed in Study 1, it seems that some practices are designed to serve the ecosystem's values, e.g., to insulate an installed base of applications from changes (Eclipse), to make it easy for end-users to install and use the latest software (R/CRAN) or to allow developers to contribute code as simply as possible (Node.js). Are particular values always associated with specific practices that further that value? We ask more generally:

RQ2.3: What is the relationship between ecosystem values and practices?

Anonymized survey data is available [7].

[20] https://cran.r-project.org/web/packages/policies.html.

5.1 Study 2 Results: Validation of Study 1

Before presenting new results from the survey, we take the opportunity to validate some of the results of Study 1, since we have available hundreds of survey responses covering similar questions from the three ecosystems in that study.

Study 1 characterized practices and values of three ecosystems based on interviews with developers in each ecosystem. The values they inferred for *Eclipse* and *Node.js/NPM* align with our data: Eclipse participants did seem to value backward compatibility as postulated: *Stability* and *compatibility* were their two highest ranked values (Table 10). Aligning with findings from the interviews, *Eclipse* developers were top-ranked in claiming to make design compromises in the name of backward compatibility (Figure 3(c)). Aligning with the interview result that showed *Node.js* developers to value ease of contributions for developers, *Node.js* participants in our survey were top ranked in valuing *innovation* and ranked highly in both making frequent changes to their own package (Figure 3(a)) and in facing breaking changes from dependencies (Figure 4(a)), although they were mid-rank in feeling any less constrained from making changes than other ecosystems (Figure 3(b)).

CRAN survey participants did not highly rank *rapid access* as expected from the interviews; and they were not more averse to adopting dependencies as predicted (not shown), although, as predicted, they did claim to clone code more (not shown). Aligning with interview results discussing personal contacts among upstream and downstream developers, they were top ranked in reporting being personally warned about changes in their dependencies (Figure 4(e)), but, contrary to expectations, were low ranked in warning their own downstream users (Figure 3(h)). This contrast, in particular, i.e., frequently being warned but rarely issuing warnings, suggests that our R/CRAN interviews may be overweighted toward downstream developers.

Although the survey largely validates the interview results, the differences highlight the fact that different methods with different sampling strategies can produce somewhat different results, and that even the design intentions of core members responsible for promulgating practices are not necessarily propagated to the whole community.

5.2 Study 2 Results: To What Extent Are Values and Practices Shared *across* Ecosystems? (RQ2.1)

The survey, policy analysis, and data mining revealed an interesting pattern of similarity and differences in values and practices across ecosystems. For those that vary across ecosystems, it is rare that we see a clear division of ecosystems in two distinct groups. Rather, sorting tends to generate a smooth curve between the extremes. Visible differences between ecosystems at either end of the spectrum are generally statistically significant, and often a few ecosystems stand out, as we will discuss. We plot answers to many of our survey questions in Figures 2, 3, and 4 and Table 7.

All values, except for *commerce* (Figure 2), were considered at least "somewhat important" in all ecosystems. *Stability*, *quality*, and *community* are nearly universal values and *compatibility*, *rapid access*, and *replicability* are also rated highly across most ecosystems (see the bottom rows of Figure 2 for the few exceptions). For *quality* in particular, participants felt even more strongly, and more consistently, that it was of high importance to them personally and to the ecosystem as a whole (the mean personal value of quality was about 0.8 scale points higher than the mean ecosystem value). Still, we see strong differences between ecosystems at each end of the spectrum. Personal values correlate strongly with perceived community values (Spearman $\rho = 0.416, p < .00001, n = 10878$, comparing the two answers for each of the eleven values, for each person, as a separate observation), but participants, on average, rated *quality* as a much higher personally,

Table 7. Comparison of Data-mined Practices (Data from libraries.io and World of Code [57]; see Section 3.3.6 for Details

	Dependency Version Constraints						
Ecosystem	(a) Exact	(b) min only	(c) range	(d) unconstrained	(e) Cloning	(f) Lock Files	(g) Maint. old vers.
Atom (plugins)	22.5%	1.55%	**73.7%**	1.29%	2.62%	0.1%	1.8%
CocoaPods	–	–	–	–	–	8.37%	3.85%
Eclipse (plugins)	–	–	–	–	–	n/a	–
Erlang,Elixir/Hex	9.09%	9.25%	**81.6 %**	0.0%	–	65.7%	3.95%
Go	–	–	–	–	3.24%	14.4% v	–
Haskell (Cabal/Hackage)	–	–	–	–	–	0.5%	1.04%
Haskell (Stack/Stackage)	–	–	–	–	–	0%	n/a
Lua/Luarocks	–	–	–	–	3.21%	0%	–
Maven	**100.0%**	0%	0%	0%	0.72% (Java)	n/a	25.4%
Node.js/NPM	16.3%	0.44%	**78.6%**	3.67%	7.03%	0.8%	3.96%
NuGet	5.27%	**88.7%**	6.01%	0%	–	7.2%	17.6%
Perl/CPAN	**100.0%**	0.0%	0.0%	0.0%	2.30%	1.0%	2.72%
PHP/Packagist	21.3%	3.72%	**66.7%**	7.99 %	1.16%	16.9%	10.6%
Python/PyPi	14.6%	34.5%	5.86%	**44.1%**	8.17%	n/a	6.07%
R/Bioconductor	–	–	–	–	3.59%	0.2%	n/a
R/CRAN	0.0%	24.4%	0.0%	**75.6%**	2.69%	0.8%	0.10%
Ruby/Rubygems	3.78%	**49.6%**	46.3%	0.94%	1.76%	17.4%	4.54%
Rust/Cargo	3.86%	2.14%	**93.6%**	.40%	6.90%	14.6%	1.4%

Dependency Version Constraints: Over all versions of packages in our data, over each of the packages' dependencies, what proportion of dependencies were constrained with *Exact* version number, specified the *minimum* version only, a *range* of versions, or left the version *unconstrained*. Dash(–) means no data (dependencies not tracked in libraries.io, or language files not indexed in WoC). Most common type of constraint for each ecosystem is **bolded**.
Cloning is percent of packages in repository whose projects borrowed a file from another package.
Maint. old vers. is percent of packages whose version number does not increase monotonically.
Lock files is percentage of packages that use a lock file to set an exact version of transitive dependencies. n/a= no equivalent of a lock file. v= Go includes projects with a "vendor" directory, which has a similar effect as a lock file.

compared to how they rated it as an ecosystem value (.9 Likert scale points, paired t-test: p<.0001); they also tended to rate *fun* slightly higher personally (.6 Likert scale, paired t-test: p<.0001); all other differences were within half a Likert scale point.

Additional values from open-ended questions. We also asked an open-ended question about other values important to their ecosystem. Common themes are counted in Table 8. Answers included *usability* (15 responses) and *social benevolence* (good conduct, altruism, empowerment, making resources available to all; 17 responses). An interesting pair of contrasting values we had not considered was *standardization* (12 responses) and *technical diversity* (17 responses). *Technical diversity* advocates valued freedom to implement things and interact with other developers in a diversity of ways: *"the package creator should be in charge of deciding how best to manage his/her package and organize with other contributors [...]"* (Node.js/NPM respondent), while *standardization* advocates said their ecosystem limited choice to save developers time and effort by promoting wide adherence to standards: e.g., a Python respondent said the platform's *"open ecosystem proposes commonly used, sensible ways to solve popular problems, enforces de facto standards"* and decried the chaos of *"NIH [Not Invented Here] syndrome."*

Table 8. Number of Respondents Suggesting Other Ecosystem Values: Usability, Social Benevolence, Standardization, Technical Diversity, Documentation, Modularity, Testability

Ecosystem	Usability	Social Benevolence	Standard-ization	Technical Diversity	Docu-mentation	Modularity	Testability
Atom (plugins)				1		1	
CocoaPods	2	2		1			
Eclipse (plugins)						1	
Erlang,Elixir/Hex	1	1	1		2		
Go	1	4	4		2	1	1
Haskell (Cabal/Hackage)						1	
Haskell (Stack/Stackage)							
Lua/Luarocks		1		1		2	
Maven			1	1			
Node.js/NPM	1	1		3		7	
NuGet	4	1					
PHP/Packagist			1				
Perl/CPAN	2	2	3	5	2	1	5
Python/PyPi	1	2	1	2	2		
R/Bioconductor		1			4		1
R/CRAN		2			1		
Ruby/Rubygems	3	3	2	2			4
Rust/Cargo		1			1	1	1
other		1	1	1		1	1

Other responses to this question we deemed to be not really ecosystem values, but rather favored technical qualities of code at the package level (64 responses), which might be promoted by ecosystem culture, such as good documentation (11 responses; 4 of which were Bioconductor participants); high modularity (16 responses; 7 of them in Node.js/NPM); and testability (11 responses; 4 each in Ruby and Perl). Finally, 13 (8%) responses objected to the framing of the question, claiming either that no community existed that could be said to share values (5 respondents, 3 of them in Maven) or saying that multiple subcommunities existed with differing values (8 respondents, including 2 in Erlang/Hex and 2 in Haskell/Cabal).

Other recent surveys [34, 77] have used similar sets of values. In light of responses to our survey, we propose the revised list of values in Appendix C. This new list adds the new values of *Standardization*, *Technical Diversity*, *Usability,* and *Social Benevolence*, removes *Quality* (since it did not distinguish among ecosystems).

Change planning practices. Participants across all ecosystems indicated in the survey (Figure 3) that they perform breaking changes only rarely: a median of *less than once a year* both for the changes that our participants perform (Figure 3(a)) and breaking changes that their package faces from dependencies (Figure 4(a)). Although prior research suggests that breaking changes are "frequent" (Section 2), this is relative to the overall frequency of change. Applying a back-of-envelope estimate to Decan et al. [21]'s findings, for example: They report about 5% of updates actually caused breakages, against a background rate of about 1.2 updates per year per package (1,029 updates to 1,710 packages in a six-month window), or one breakage every 17 years. Given that breakages may not be evenly distributed, packages have multiple, recursive dependencies, and developers work on multiple packages, experiencing a breakage once a year is in the range of

Table 9. Comparison of Sanctioned Practices and Features

Ecosystem	(a) Dependencies outside repository	(b) Central Repository	(c) Access to old dependency versions	(e) Gatekeeping standards	(f) Synced ecosystem
Atom (plugins)	●	●	●	○	○
CocoaPods	●	●	●	○	○
Eclipse (plugins)	●	○	●	*n/a*	● core
Erlang,Elixir/Hex	●	●	●	○	○
Go	●	○	○ w/extra work	*n/a*	○
Haskell (Cabal/Hackage)	● alt repo	●	●	● submitter	○
Haskell (Stack/Stackage)	○	●	◑	● compatibility	○
Lua/Luarocks	○	●	●	○	○
Maven	●	●	●	○	○
Node.js/NPM	●	●	●	○	○
NuGet	● alt repo	●	●	● Virus-free	○
Perl/CPAN	● alt repo	●	●	◑ staged releases	◑ staged releases
PHP/Packagist	●	●	●	○	○
Python/PyPi	●	●	●	○	○
R/Bioconductor	● alt repo	●	◑	◑ staged releases	◑ staged releases
R/CRAN	● alt repo	●	○ w/extra work	●	○
Ruby/Rubygems	●	●	●	○	○
Rust/Cargo	●	●	●	○	○

● = ecosystem has feature, ○ = does not have feature, ◑ = has feature, but for a group of packages, not for individual packages. *alt repo* = through reference to an alternative repository; *staged releases* = groups of packages are debugged together and released as a group. *submitter* = the author, not the package, is vetted. *core* = core packages only. See Section 3.3.5 for details.

plausability. So, this is perhaps why their actual experience of dealing with a breaking change may be infrequent even if breaking changes are frequent overall in the ecosystem.

Respondents in every ecosystem agreed, on average, that they used semantic versioning or comparable versioning strategies (Figure 3(f)), batch multiple changes into a single release (Figure 3(d)), document their changes (Figure 3(e)), and are conservative about adding dependencies to their projects (Figure 4(c)). These seem to generally be considered as good software-engineering practices independent of programming language or ecosystem.

Answers that varied more dramatically among ecosystems included reluctance to make breaking changes (Figure 3(b)), willingness to compromise design for backward compatibility (Figure 3(c)), and synchronizing with users before releasing changes (Figure 3(h)). Data mining reveals that ecosystems also vary considerably in how often they make updates to previous versions, ranging from as high as 25% of Maven projects doing this at least once, to 0.1% of R/CRAN projects doing so.

Turning to shared community resources, all but two of the ecosystems we studied supply a central repository server from which packages could be downloaded automatically as needed (Table 9(b)). Two (Go and Eclipse) only maintain indexes to maintainers' own servers that must supply the package and metadata in some standard way. Advertised submission requirements for packages show that ecosystems differed in the level of vetting (Table 9(e)) of the packages

these repositories apply. Haskell's Cabal/Hackage system is unusual in that it vets maintainers, who apply for accounts that are hand-checked by human reviewers, but does not apply more than minimal automated standards to submitted packages. CRAN has very strict standards for package submissions and updates,[21] which are vetted by hand as well as automated tests.

Three ecosystems are released all at once on a regular, synchronized schedule (Table 9(f)): the core set of packages in Eclipse, as well as the whole of Bioconductor (synchronized with releases of the R runtime), and CPAN. These work by having a staged sequence where a development build is worked on until it is consistent, then parts or all of it are released as a group into the official supported release. Other ecosystems allow developers to release packages whenever their authors wish. This is similar to practices of operating-system-level software ecosystems such as Debian's APT[22] that repackage software from a variety of languages and ecosystems into compatible releases for an operating system.

Note that Stackage's sets of compatible packages are curated together post hoc[23]; their development is not synchronized unless developers collaborate on their own to do so.

Practices for coping with dependency changes. Sixteen of the 18 ecosystems offer an optional (Table 9(b)) but widely used central repository (Table 9(a)) for packages, usually encouraging packages to refer to dependencies by name and version number.

When asked specifically about their package's exposure to breaking changes from upstream packages, participants across all ecosystems again reported low frequencies (Figure 4(a)); only a quarter of our participants indicated that they saw a breaking change per year. Participants in ecosystems with more conservative change practices (e.g., *Eclipse, Erlang, Perl*) are exposed to slightly fewer breaking changes. Participants across all ecosystems indicated that they are conservative in adding dependencies (Figure 4(c)) and perform significant research first (Figure 4(d)). In contrast, how they learn about updates (Figures 4(e)–(g); e.g., through personal contacts or tools), the rate to which they may skip them (Figure 4(h)), and how they declare version constraints on dependencies (Figure 4(i)) depends significantly on the ecosystem.

Data mining (Table 7) reveals that file cloning is rare (less than 10% of projects) in every ecosystems in which we measured it; developers instead rely on the package dependency infrastructure (Table 7(e)). Mining also confirmed survey answers about how users of packages chose to constrain the versions of packages they depended on: While Maven almost universally relies on a fixed version number (e.g., package A might depend on precisely version 3.2.1 of package B), other ecosystems typically constrain dependencies to version number ranges (Node.js/NPM, Atom, PHP, and Rust/Cargo), specifying only a minimum version (NuGet, Ruby/RubyGems) or leaving versions unconstrained (Python/PyPi, R/CRAN). Survey and mining results differed for one ecosystem, however: Perl/CPAN users claimed the ecosystem's typical practice was to specify just the name (43% of respondents) or version range (36%) of dependencies, yet mining of libraries.io revealed nearly 100% use of exact version numbers. This may be a matter of developer perception: libraries.io apparently measures precise dependencies captured in the published repository, but tools such as `Dist::Zilla::Plugin::DistINI` generate these from less-constrained numbers specified by developers.

Universal or distinctive. While there is considerable nuance in the differences among ecosystems, overall our results suggest that there are several values that seem to be universal, at least

[21] https://cran.r-project.org/web/packages/policies.html.
[22] https://wiki.debian.org/Apt.
[23] https://github.com/commercialhaskell/stackage#frequently asked-questions.

in the 18 ecosystems we surveyed. Chief among these are stability, quality, and community, while compatibility, rapid access, and replicability have achieved a near-universal status. The unique personality of each ecosystem, however, seems to derive from either a few key distinctions (in values or in practices) that set them apart. There are many examples of this, including:

- *Bioconductor* and *Eclipse* stand out as coordinating releases on a synchronized and fixed schedule and the survey (Figures 3(i) and (j), Table 9(f)) and valuing *curation* (Figure 2, Table 9(e)).
- *Go* has a distinctive version numbering practice that does not require version updates on all changes (Figure 3(g), Table 9(c)).
- *CRAN* and *Bioconductor* have strict requirements for submission and update of packages (Figure 3(k), Table 9(e)).
- *Lua* developers value *fun*, feel least constrained from making changes in their code, and generally do not coordinate much with others (Figures 3(b),(h), and (i)).
- *Rust* has a strong stance on *openness* and is the least prone to make design compromises for backward compatibility (Figure 3(b) and (c)). Data mining of Cargo projects show they rarely port fixes to earlier code releases (Table 7(g)).
- *CPAN* developers universally claim to write change logs (Figure 3(e)).

Value differences by ecosystem are statistically significant for each of the values (Kruskal-Wallis, run separately on each value to check if it differs by ecosystem: $p<.00001$, χ^2 ranging from 53.704 for *quality* to 178.69 for *commerce*).

Summary of RQ2.1 results: Stability, quality, community, compatibility, rapid access, and replicability are important across all ecosystems, while openness, curation, standardization, technical diversity are values that are not universal, but differ by ecosystem. Breaking changes are experienced only rarely by any one developer (on the order of yearly), even though they are common within an ecosystem as a whole. Differing ecosystem circumstances lead to great variety in developers' willingness to make breaking changes, or conversely to compromise their designs to ensure backward compatibility; and in turn consumers' eagerness to incorporate upstream changes.

5.3 Study 2 Results: To What Extent Is There a Consensus *within* Ecosystems about Values and Practices? (RQ2.2)

The distribution of value ratings *within* each ecosystem was particularly wide for the values *replicability*, *openness*, and *curation*, indicating generally less consensus on these values. There is evidence of broad consensus about the highest ranked value(s) for some ecosystems (Table 10), most conspicuously in cases in which a value clearly aligns with the core purpose of an ecosystem. An illustrative example is Stackage and Cabal/Hackage, two Haskell-based ecosystems, contrasted strongly with each other in *compatibility* and *curation*; participants rated these values as much more important in Stackage than in Hackage/Cabal. Stackage was also rated markedly lower in *rapid access* than all other ecosystems. These values are consistent with the stated goals of Stackage ("to create stable builds of complete package sets"). Stackage is built on top of Cabal for the express purpose of curating compatible sets of versions, while Hackage submissions only require that they be submitted by a developer whose identity has been manually vetted (Table 9(e)). Volunteer curators wait until a set of consistent package versions can be assembled and release them as

Table 10. Values Most Commonly Rated Highest, by Ecosystem

Ecosystem	Top 3 values	Consensus in % C1	C2	C3
Haskell/Stack	compatibility > replicability > curation	75	55	45
Perl/CPAN	stability > replicability > quality	64	40	31
Maven	replicability > stability > quality	64	38	32
Lua/Luarocks	fun > replicability > quality	64	35	17
Ecilpse	stability > compatibility > quality	62	48	37
NuGet	replicability > compatibility > stability	59	37	20
Go	quality > stability > fun	56	37	19
R/Bioconductor	replicability > quality > compatibility	52	32	26
CocoaPods	quality > stability > compatibility	52	30	17
Rust/Cargo	replicability > stability > community	51	31	23
PHP/Packagist	quality > stability > compatibility	50	32	23
Node/NPM	rapid.access > community > innovation	50	24	15
Atom	rapid.access > fun > openness	50	26	17
Erlang	quality > fun > stability	46	24	18
Haskell/Cabal	quality > innovation > replicability	43	17	8
Python	replicability > quality > stability	42	20	14
Ruby	fun = community = rapid.access	41	18	12
R/CRAN	replicability > compatibility > innovation	36	20	8

Consensus C*n* is the percent of respondents in each ecosystem who did not rate any value higher than any of the ecosystem's highest *n* values. Top three values are listed for each ecosysem; > indicates relative popularity of the values; = indicates ties.

a unit, trading rapid release for tested compatibility. The Stackage/Hackage choice is controversial in the Haskell community, which may make their perceived differences in values and practices more visible.

A few more examples include:

- *Maven* is primarily a build tool that comes with a centralized hosting platform for Java packages and was not designed as a collaborative platform. This purpose is reflected in strongly valuing *replicability* but least valuing *community*, *openness*, or *fun*.
- *Bioconductor* is a platform for scientific computation (specifically, analysis of genomic data in molecular biology) where *replicability* of research results is a key asset, but *commerce* is clearly not a focus.
- *Lua* is widely used as an embedded scripting language for games; prior work has shown that the culture of game developers is significantly different from that of application developers [58]; for example, game development communities value creativity and communication with designers over rigid specifications, which makes extensive automated testing impractical.

Others, like *R/CRAN*, have markedly less consensus, at least regarding the set of values that we surveyed.

Some, but not all, practice differences can be explained by enforced policies or design choices in platform tools. For example, *Node.js/npm* sets a version range for dependencies by default when a dependency is added (Figure 4(i)), *Bioconductor* and the core packages of *Eclipse* have a

synchronized, central release (Figure 3(i) and (j), Table 9(f)), and *Bioconductor* and *CRAN* require reviews before packages are included in the repository (Figure 3(k), Table 9(e)). Some practices are supported by optional tooling in the ecosystem, such as tools to create notifications on dependency updates in the *Node.js* and *Ruby* community (Figure 4(i); e.g., *gemnasium* and *greenkeeper.io*). Other practices seem to be mere community conventions—for example, providing change logs is encouraged in the documentation of *CPAN* but not enforced, yet the practice is apparently universal (Figure 3(e)).

Interestingly, there are some cases of practices with surprisingly little consensus in some ecosystems given what we know about tools and policies in that ecosystem. For example, 26.6% of *Node.js* respondents indicated that a "package has to a meet strict standards to be accepted into the repository" (Figure 3(k)), even though that community's *npm* repository does not have any such checks (Table 9(e)) and in fact contains many junk packages. It may be that ecosystem members are not aware of the design space and what practices other ecosystems employ, so they have a biased interpretation of what a "strict standard" is. Alternatively, participants may be members in subcommunities with contrasting values and practices. For example, there may be vetting of revisions among the developers within a specific project or subcommunity that is also hosted on *npm*.

The role of roles. We wanted to explore the possibility that survey respondents' differences in perceived values and practices may be explained by the role of a respondent in their ecosystem. The ecosystem may appear different depending on one's responsibilities and perspective. The survey asked people what their role was in the ecosystem: choices were *user*, *committer*, *submitter*, package *lead*, central package lead (a.k.a. *lead+*), and *founder*. We analyzed how core (*lead+* and *founder*) roles differed from the rest within each ecosystem. We suspected that core and peripheral ecosystem participants may have different values, but we found little evidence that that was the case. We tested their ratings on the perceptions of all 11 values and found that only for one value, replicability, was there a statistically significant difference (t-test, p=0.044, n=1,504); however, this difference was small (an average rating 3.5 out of 5 for core, 3.68 for non-core, thus a difference of 0.18 scale points), and there was no evidence that value perceptions differed for other values (t-test, p between .13 and .73, *n* ranging from 1,492 to 1,504).

Core people seemed to be more enmeshed in the community than the other roles, in the sense that they were more likely to collaborate with upstream packages (χ^2 (1, N=932) = 16.571, p < .0001; 21% more likely to answer yes to the question, "In the last 6 months I have participated in discussions, or made bug/feature requests, or worked on development of another package in <ecosystem> that one of my packages depends on.") have downstream dependencies (χ^2(1, N=925) = 24.132 p < .0001, 18% more likely to answer yes to the question, "Have you contributed code to an upstream dependency of one of your packages in the last 6 months (one where you're not the primary developer)?"), and claim to know their users' needs (χ^2(1, N=932) = 62.947 p < .0001, 29% more likely to answer "Strongly" or "Somewhat agree" to the question, "I know what changes users of <package> want"). People in core roles felt very slightly more confident in their answers to the community values questions, (χ^2(1,N=932) = 6.2247 p <.05, 8% more likely to answer "Confident" or "Very confident" to the question "How confident are you in your ratings of the values of <ecosystem> above?"); this difference was statistically significant, but not very large.

In short, there are a few features that distinguish core community members from the rest, but they seem to be culturally a part of their communities in that they perceive its values to be the same.

Summary of RQ2.2 results: Ecosystems tend to have many of the same values but distinguish themselves by virtue of a few distinctive values strongly related to their purpose and audience. Consensus in practices is largely, but not entirely, driven by the affordances of shared tooling and the policies that they enforce or encourage. Core and peripheral members of the ecosystem community share their ecosystem's values, but core members are more collaborative in their practices.

5.4 Study 2 Results: What Is the Relationship between Values and Practices: The Case of Stability (RQ2.3)

One might expect that ecosystems that share similar values would adopt similar practices that support those values, but for most practices that is not the case. We averaged each value and practice answer within each ecosystem to get a summary for each ecosystem of mean answers and looked for correlations between any value and any practice among columns within these 18 rows. There were few strong correlations between values and practices. Out of 418 such value-practice comparisons, only 29 were significantly correlated (Spearman test, $p<0.05$); however, even these may be due to chance: Because of the small sample size (n=18) and the large number of comparisons, applying a Holm-Bonferroni correction rules out taking any of these correlations as conclusive.

The fact that practices are not universally associated with particular values implies that the same value can be associated with the adoption of different practices. For example, of the practices shown in the violin plots above,[24] only one, the perception of the ecosystem's use of exact version numbers to refer to dependencies (Figure 4(i), choice E), significantly correlated with the perceived value of stability to the ecosystem (Spearman correlation of mean answers within each ecosystem : $\rho = 0.506, p < .05, n = 18$ ecosystems). We investigate further this relationship with a comparison of the practices associated with stability in three ecosystems that had high ratings and high consensus for stability: *Eclipse, Perl,* and *Rust* (Figure 2 and Table 10). Our survey results indicate that these ecosystems achieved stability with different, sometimes nearly opposite, practices.

- *Eclipse: stability through strict standards and gatekeeping. Eclipse*'s leadership very strongly promotes stable plugin APIs. As we mentioned earlier, official developer documentation includes this *"prime directive": "When evolving the Component API from release to release, do not break existing Clients"* [25]. Eclipse developers rated *stability* higher than any other ecosystem, and with the smallest variance in their mean ratings of stability (Figure 2), and strong consensus that stability was the highest value (cf. Table 10).

  Survey answers about practices show that *Eclipse* relies on gatekeeping (Figure 3(k)) and its developers claim to make design compromises to achieve backward compatibility (Figure 3(c)); they police each others' backward compatibility and release together when they can be sure they will not break legacy code (Figure 3(i)); developers feel constrained in making changes (Figure 3(b)).
- *Rust: stability through dependency versioning and stability attributes. Rust*, in contrast, ranked lowest in design compromises for backward compatibility (Figure 3(c)) and rarely maintains outdated versions, (Table 7(g)), but is high in semantic versioning (Figure 3(f)). Rust's *Cargo* infrastructure prevents the use of wildcards for dependency versions, although it allows ranges (Figure 4(i)), which are almost universally used (93.6% of Cargo packages, Table 7(c)).

[24]Figures 3(b)–(k) and Figures 4(c)–(h) and (j), and the four answers of Figure 4(i) taken separately.

Users were thus prodded to use older versions of dependencies, rather than letting their tools upgrade them automatically and burdening upstream packages with bug reports when things change. Other stability features include a "lock" file that records exact versions of dependencies used by a version (Table 7(f)), and a feature called "stability attributes," which tag API elements that are guaranteed to be stable, in contrast to new features that might change [80].

Survey results show that *Rust* developers acknowledged the community's stated value of *stability* (Figure 2), despite the fact that participants also perceived the ecosystem's packages to be in fact relatively unstable (Figure 4(b)). The *Rust* language developers had been consistent in promising stability for the "stable" branch of the language, to the extent that they test any compiler changes against the entire corpus of *Rust* programs they can find on GitHub. But their analysis of their community's 2016 user survey [79] summarized why many users complained about instability: too many packages ("crates") relied on unstable "nightly" development versions of the compiler to take advantage of interesting new features. They concluded that *"consensus formed around the need to move the ecosystem onto the stable language and away from requiring the nightly builds of the compiler."*

- *CPAN: stability through centralized testing*. Finally, *Perl*, unlike *Rust*, is low in semantic versioning (Figure 3(f)), and in fact was the most likely ecosystem to claim they refer to dependencies by name only, not version number (Figure 4(i)). They indicate some gatekeeping and design compromises but not to the extent of Eclipse (Figure 3(c) and (k)). However, in response to the open-ended question about what other values were not covered by the survey, 12 (40%) of 30 Perl/CPAN participants who gave comments mentioned *testability*,[25] many referring to Perl's extensive battery of tests run on CPAN packages by volunteers; one explicitly claimed this test facility helped with the stability of Perl packages. CPAN stages changes and releases packages together (Table 9(f)), almost entirely specifying fixed version numbers of their dependencies (Table 7(a)). A Haskell/Hackage participant mentioned CPAN's *kwalitee* metric, an operationalization of quality employed by these testing facilities, and attributed it to the ecosystem's "focus on stability and compatibility."

The three ecosystems work towards stability in very different ways. Eclipse, with its long-standing corporate support, is able to dictate that upstream developers pay the cost of maintaining backward compatibility; Rust/Cargo, although users clamor for stability, is eager to attract developers, and cannot impose the cost of stability by fiat as in Eclipse; instead, they apply gentle pressure to upstream developers in various ways, while *easing* the pressure from downstream developers by discouraging automatic major updates. CPAN, finally, has a large cadre of volunteers (CPAN Testers) and built infrastructure taking on the task of thorough testing.

This comparison of stability practices demonstrates that the relationships between practices and values are context-dependent and thus hard to generalize. A comprehensive theory incorporating such insights is a task for future work. We hope our dataset and the questions it suggests provide a useful launching point. Contrasts revealed by the survey are ripe for further investigation: researchers can find appropriate subjects for case studies of values being pursued in contrasting ways, or, conversely, practices associated with contrasting values. In this case, analyzing the differences between these three ecosystems suggests that the theory of how practices can further values should take into account other factors, including the presence, availability, and motivations of different kinds of developers. This should be confirmed, however, with more exhaustive study of

[25] *Testability* was not a value we surveyed, but we recommend it as a new value in an expanded list, since many survey takers suggested it.

these and other ecosystems and with other practice contrasts. Ecosystem communities dissatisfied with their practices can themselves use it as a starting place to find alternative combinations of practices that others are using.

Summary of RQ2.3 results: Many ecosystems have clear distinctions in a few key values and practices. Often the consensus on important values is high; some practices are actually enforced by policies and platform tools. However, some values, particularly *quality*, are nearly universal value for software engineers with little variance among ecosystems. Breaking changes are also generally avoided, though the strategies of how this is achieved and as how difficult it is perceived to be depends on the specifics of the ecosystem.

6 DISCUSSION AND FUTURE WORK

Our article makes several contributions toward understanding how ecosystems go about the critical task of managing breaking changes and how those practices reflect the culture and values of the ecosystem participants. Study 1 contributes a qualitative accounting of the very different ways that three contrasting ecosystems manage change and how these differences relate to different values and different ideas about which classes of participants should bear the costs. Prior work [19, 36, 67, 72] has examined particular practices for change management and noted the prevalence of breaking changes [22, 48, 54, 90]. Our contribution is to characterize the types of change negotiation practices found in three different ecosystems, show how these different sets of practices require varying amounts of effort from different classes of ecosystem participants. We also show how these different sets of practices reflect ecosystem values about the software, the community, and which community needs take precedence. Study 2 builds on this, examining practices and values in a larger set of 18 ecosystems. We find that some values appear to be universal or nearly so, within this set of ecosystems, perhaps reflecting a broader open source culture. Other values show considerable divergence, which appears to be a substantial component of ecosystems' distinctive "personalities." Within ecosystems, some values appear to reflect a consensus among participants, while views of others are highly variable, perhaps reflecting diverse views of subsets of projects or individuals, rather than ecosystem-wide values. We also show that the relationship between practices and values is not simple, and we illustrate the apparent nature of such relationships by contrasting the very different practices that several ecosystems employ in pursuit of stability, which all of them value highly.

In the following subsections, we outline new and interesting research questions brought to light by this work.

6.1 When Are Practices in Conflict or Complementary?

It seems highly unlikely that practices can be treated as independent of one another. If an ecosystem is considering adopting a new practice, e.g., to enhance stability, the outcome of trying to implement various stability-enhancing practices is likely to be contingent on the set of other practices already in place. For example, introducing semantic versioning to signal breaking changes would not make sense where snapshot consistency (current versions of everything must be compatible) is already enforced. Complementarity is the other side of the coin: Certain practices may be more effective if certain other practices are adopted as well. For example, centralized testing is likely to be more effective where an ecosystem has a repository with strong gatekeeping mechanism, and a norm that dissuades developers from using alternative repositories.

We suspect that many conflicts and complementarities among practices are much more subtle, and greater insight into these relations among practices would be very helpful to clarifying feasible

paths for achieving ecosystem goals. Our survey data contains many starting points for investigations; for example, by allowing researchers to identify ecosystems with various combinations of values and practices as targets for further exploration.

6.2 Assimilation or Ecosystem Selection?

Our survey indicates that developers' personal values usually align well with the values of ecosystems (Figure 2) in which they operate. Understanding how this alignment comes about would help to predict the outcome of attempted interventions and design interventions more likely to be effective. There are at least two major possibilities. Developers may join ecosystems for reasons unrelated to values, e.g., the application domain or technical characteristics of the software. Being exposed to the ecosystem values, they may then assimilate over time, adapting their behavior and personal values to what they experience around them. However, the alignment may come about primarily through value-based selection, where developers join ecosystems because they resonate with the system's values.

These two possibilities will often carry different implications for interventions. If developers tend to assimilate the ecosystem's values, an existing community might be steered toward different practices and expect that developers will adapt over time. In contrast, if developers pick ecosystems based on compatible values, then this would likely mean that substantial changes would attract new value-aligned developers but risk significant disruption if long-term contributors rebel or leave. While one might expect some degree of both selection and assimilation, understanding which values and practices are more easily adapted, and which tend to be resistant to change, could be a big help in designing effective interventions.

Our survey data does not provide insights into causation, but it can provide starting points for further investigations and can be combined with external data to approach the questions. We took a small step in this direction to illustrate some of the possibilities. If developers tend to assimilate practices and values from those around them, we would expect values and practices to be shared more among ecosystems with relatively large overlap of participating developers than in those with a relatively small overlap. As a preliminary study, we investigated whether ecosystems that share many developers[26] have similar practices or values. Over all pairs of ecosystems, we found a sizable correlation between similarity of average responses on ecosystem *practice* questions (those depicted in Figures 3 and 4), and overlap in committers to those ecosystems (Spearman $\rho = 0.341, p < .00001, n = 289$ pairs of ecosystems, correlating average perceived ecosystem value for each pair of ecosystems with developer overlap between them). Interestingly, perceived *values* of the ecosystem do *not* seem to align with developer overlap ($\rho = -0.05, p = 0.44, n = 289$, correlating average personal value for each pair of ecosystems with developer overlap between them).

While a number of interpretations of these relationships are possible, the data are consistent with the idea that practices diffuse among ecosystems that have large developer overlap, but values do not. Future work using time series data about developer overlap and historic participation in ecosystems would allow researchers to identify specific developers that moved to ecosystems with different or similar practices and values (according to our survey data) and use interviews, surveys, or data mining to see if and how their behavior changed.

[26] To measure developer overlap, we assembled a list of all packages in each ecosystem from libraries.io, Cargo.io, and LuaRocks.com, and we identified Eclipse plugins as non-fork packages in GitHub containing a "plugin.xml" file. Using the authors of commits to those packages' github projects as archived by Mockus [57], we counted what percent of each ecosystem's contributors also contributed to each other ecosystem. We excluded Bioconductor, because we had no clear mapping to GitHub repositories.

6.3 When Are Attempted Changes Broadly Adopted?

Collecting cases of effective and ineffective past changes in ecosystems can help to understand the conditions that favor broadly adopted changes. Examples of attempted policy or practices changes can often be found through surveys. In our survey, text answers about contrasting ecosystems often explained how practices were deliberately designed. Five Perl developers, for example, described how an extensive centralized testing infrastructure (CPAN Testers) was added to improve the quality and compatibility of CPAN modules. Perhaps beginning with our results and then conducting new interviews or surveys, it should be possible to unearth many examples of attempted change and to determine the outcome. A second approach could identify conflicts between values and practices to suggest ineffective changes. In the case of Rust, for example, the high value of stability (Figure 3(a)), but also high perception of instability (Figure 4(b)) led us to investigate Rust's struggle, as mentioned above, to promote practices leading to stable versions of libraries despite the community's eagerness to innovate with new features.

In Edgar Schein's work on organizational culture, his recommendations [70, p. 323ff] for changing an organization include strong role models for new behaviors, lowering learning anxiety, and raising survival anxiety (i.e., making people confident that they can learn new practices and aware that the community will fail if they do not). Elements of this advice are visible in the practices of ecosystems that have tried to change their values. In Rust, for example, the compiler team models stability practices that packages might follow [80]. Rust's stability attributes for packages may reduce learning anxiety by making it easier for downstream users to create stable interfaces, and Rust's annual survey helps developers see each others' agreement that there are problems with stability.

7 CONCLUSION

While managing change has long been an important topic in software engineering, it is particularly interesting in the context of open source ecosystems, since projects tend to be highly interdependent yet independently maintained. The variety of practices used to manage change is considerable, but perhaps most interestingly is what we might think of as the political dimension in the selection of practices. Whose interests are served by the adoption of one set of practices rather than others? How are the costs (primarily effort) distributed over types of ecosystem participants? What values to these practices actually serve?

We have attempted to provide a somewhat detailed description of practices used in three ecosystems, as well as a broader characterization of 18 ecosystems. We believe these studies just scratch the surface, however, and much work remains to be done in understanding how practices fit with values, and with each other, and how effective changes can be made to address ecosystem weaknesses. We hope through this work, and through the data we are making publicly available, to have contributed to a better understanding of these issues.

APPENDICES

A STUDY 1 INTERVIEW PROTOCOL

The following lists the questions from our interview script. We did not ask each question to each interviewee, but instead we directed them towards areas where they had personal experience. Given our iterative approach, some questions in this script were added or modified after earlier interviews.

For maintainers of upstream packages:

- Why do you work on <package1>?
- Do you have any plan or strategy for how the interface of <package1> will evolve as people come to depend on it?

- Think about a recent larger change in your project. Was it backward-compatible? What impact did you expect it would have on packages that depend on <package1>?
- Follow up: Did you consider alternative ways of making <change1> that would have more or less impact on users of <package1>?
- Follow up: If you had not made <change1>, what would have happened differently for <package1>'s future?
- Follow up: What is your position on backward compatibility?
- Does the platform help/hinder you in evolution decisions as in <change1>? What if the platform had mechanism <alternative mechanism>?

For developers with upstream dependencies:

- Why do you work on <package1>?
- If there's a useful looking package that claims to provide some functionality you need, how do you decide whether to adopt it?
- What's your general strategy for choosing which version of a package to depend on?
- When do you think it's reasonable and expected for a package to change its interface?
- Do you prefer a stable but stale or a rapidly evolving but unstable dependency? What rate of interface change is too often?
- Is it a burden to have too many dependencies for a project?
- Can you give an example of a package you've considered, and felt like its stability was a consideration (positively or negatively)?
- How do you keep up with changes to packages you depend on?
- When <change1> happened in <upstream package1>, how did you first find out about it?
- Are you ever watching for development activity between releases?
- Are you using the Github notification mechanism and why/why not?
- If you could have an ideal notification system to get important changes: What would such system look like, what changes would it notify you about?
- Did you think <change1> was an appropriate change, or should they have left it alone?

For developers having experience working on the platform, we asked questions about specific policies, their intentions, and their consequences. Here are some example questions about CRAN:

- CRAN differs from some other repositories in that it asks package authors to notify reverse dependency packages before submitting an update that breaks its API.
  - —Was there anything specific that precipitated that policy?
  - —Did you consider other options for solving the problem? What were the tradeoffs you thought about?
  - —How successful has that policy been so far?
- More generally, CRAN has stricter requirements for authors than some other package repositories do. What factors does the CRAN team take into consideration when deciding if a quality standard is worth the effort of instituting and enforcing?
- Bioconductor does coordinated releases of all the packages at once, while CRAN lets packages update on their own schedule.
  - —How and why did the two repositories end up having different policies?
  - —What have been the consequences for the two repositories?
  - —Will they likely stay that way?
- CRAN makes it easy to install only the latest version of a package; some repositories let users install old versions. Why is it done that way?

- CRAN has more permissive expectations about version number changes than some platforms. Has the current system been sufficient, or have you considered altering the policies about numbering?
- Can you tell me something about how potential breaking changes are handled among the developers of the base and recommended packages?
  —How do developers communicate to coordinate and synchronize changes?
  —Does it work differently for base and recommended than among ordinary packages in the CRAN repository?

B STUDY 2 SURVEY QUESTIONS

For transparency and replicability, we list all evaluated questions of the survey including their exact phrasing. We exclude a small number of questions about power structures, community health, and motivation that we have not used in this article.

Part I: Ecosystem.

- Please choose ONE software ecosystem* in which you publish a package**. If you don't publish any packages, then pick an ecosystem whose packages you use.
  "Software ecosystem" = a community of people using and developing packages that can depend on each other, using some shared language or platform
  * "Package": A distributable, separately maintained unit of software. Some ecosystems have other names for them, such as "libraries," "modules," "crates," "cocoapods," "rocks," or "goodies," but we'll use "package" for consistency.
  [selection or textfield, substituted for <ecosystem> in remainder of survey]

Ecosystem Role.

- Check the statement that best describes your role in this ecosystem.
  —I'm a founder or core contributor to <ecosystem> (i.e., its language, platform, or repository).
  —I'm a lead maintainer of a commonly-used package in <ecosystem>.
  —I'm a lead maintainer of at least one package in <ecosystem>.
  —I have commit access to at least one package in <ecosystem>.
  —I have submitted a patch or pull request to a package in <ecosystem>.
  —I have used packages from <ecosystem> for code or scripts I've written.
- About how many years have you been using <ecosystem> in any way?
  —< 1 year
  —1–2 years
  —2–5 years
  —5–10 years
  —10–20 years
  —> 20 years

Ecosystem values.

- How important do you think the following values are to the <ecosystem> community? (Not to you personally; we'll ask that separately.) [See Section 3.3.2 for the 11 value questions; **results shown in Figure 2.**]
- How confident are you in your ratings of the values of <ecosystem> above?
  —Not confident
  —Slightly confident

—Confident
—Very confident
- Is there some other value the <ecosystem> community emphasizes that was not asked above? If so, describe it here:

Part II: Package.

- In the following, we are going to ask about your experience working on one particular package. Please think of one package in <ecosystem> you have contributed to recently and are most familiar with. If you haven't contributed to a package in <ecosystem>, then name some software you've written that relies on packages in <ecosystem> packages. You may use a pseudonym for it if you are concerned about keeping your responses anonymous. — [text fields, substituted for <package> in remainder of survey]
- Do you submit the package you chose to a/the repository associated with <ecosystem>? (Choose "no" if the ecosystem does not have its own central repository.) — [yes/no]
- Is there any software maintained by other people that depends on the package you chose? — [yes/no]
- Is the package you chose installed by default as part of a standard basic set of packages or platform tools? — [yes/no]
- How important are each of these values in development of <package> to you personally? **[See Section 3.3.2 for the 11 value questions.]**
- (OPTIONAL) Is there some other value important to you personally for <package> which was not mentioned? — [text fields]
- How often do you face breaking changes from any upstream dependencies (that require rework in <package>)? **[Results shown in Figure 4(a).]**
  —Never
  —Less than once a year
  —Several times a year
  —Several times a month
  —Several times a week
  —Several times a day
- How often do you make breaking changes to <package>? (i.e., changes that might require end-users or downstream packages to change their code) — [frequency scale as above]**[Results shown in Figure 3(a).]**

Making changes to <package>.

- I feel constrained not to make too many changes to <package> because of
- potential impact on users. **[Results shown in Figure 3(b).]**
  —Strongly agree
  —Somewhat agree
  —Neither agree nor disagree
  —Somewhat disagree
  —Strongly disagree
  —I don't know
- I know what changes users of <package> want. — [agreement+don't know scale as above]
- If I have multiple breaking changes to make to <package>, I try to batch them up into a single release. — [agreement+don't know scale as above]**[Results shown in Figure 3(d).]**
- I release <package> on a fixed schedule, which <package> users are aware of. — [agreement+don't know scale as above]**[Results shown in Figure 3(j).]**

- Releases of <package> are coordinated or synchronized with releases of packages by other authors. — [agreement+don't know scale as above]**[Results shown in Figure 3(i).]**
- When working on <package>, I make technical compromises to maintain backward compatibility for users. — [agreement+don't know scale as above]**[Results shown in Figure 3(c).]**
- When working on <package>, I often spend extra time working on extra code aimed at backward compatibility. (e.g., maintaining deprecated or outdated methods) — [agreement+don't know scale as above]
- When working on <package>, I spend extra time backporting changes, i.e., making similar fixes to prior releases of the code, for backward compatibility. — [agreement+don't know scale as above]

Releasing Packages.

- A large part of the community releases updates/revisions to packages together at the same time. — [agreement+don't know scale as above]
- A package has to a meet strict standards to be accepted into the repository. — [agreement+don't know scale as above]**[Results shown in Figure 3(k).]**
- Most packages in <ecosystem> will sometimes have small updates without changing the version number at all. — [agreement+don't know scale as above]
- Most packages in <ecosystem> with version greater than 1.0.0 increment the leftmost digit of the version number if the change might break downstream code. — [agreement+don't know scale as above]
- I sometimes release small updates of <package> to users without changing the version number at all. — [agreement scale, without "don't know"]**[Results shown in Figure 3(g).]**
- For my packages whose version is greater than 1.0.0, I always increment the leftmost digit if a change might break downstream code (semantic versioning). — [agreement as above]**[Results shown in Figure 3(f).]**
- When making a change to <package>, I usually write up an explanation of what changed and why (a change log). — [agreement as above]**[Results shown in Figure 3(e).]**
- When working on <package>, I usually communicate with users before performing a change, to get feedback or alert them to the upcoming change. — [agreement as above]**[Results shown in Figure 3(h).]**
- When making a breaking change on <package>, I usually create a migration guide to explain how to upgrade. — [agreement as above]
- After making a breaking change to <package>, I usually assist one or more users individually to upgrade. (e.g., reaching out to affected users, submitting patches/pull requests, offering help) — [agreement as above]

Part IV: Dependencies.

- In the last 6 months I have participated in discussions, or made bug/feature requests, or worked on development of another package in <ecosystem> that one of my packages depends on. — [yes/no]
- Have you contributed code to an upstream dependency of one of your packages in the last 6 months (one where you're not the primary developer)? — [yes/no]
- About how often do you communicate with developers of packages you depend on (e.g., participating in mailing lists, conferences, Twitter conversations, filing bug reports or feature requests, etc.)? — [frequency scale, as above]**[Results shown in Figure 4(f).]**

For most dependencies that my packages rely on, the way I typically become aware of a change to the dependency that might break my package is:

- I read about it in the dependency project's internal media (e.g., dev mailing lists, not general public announcements) — [agreement scale, as above]
- I read about it in the dependency project's external media (e.g., a general announcement list, blog, Twitter, etc) — [agreement scale, as above]
- A developer typically contacts me personally to bring the change to my attention — [agreement scale, as above]**[Results shown in Figure 4(e).]**
- Typically I get a notification from a tool when a new version of the dependency is likely to break my package — [agreement scale, as above]**[Results shown in Figure 4(f).]**
- Typically, I find out that a dependency changed because something breaks when I try to build my package. — [agreement scale, as above]**[Results shown in Figure 4(g).]**
- How do you typically declare the version numbers of packages that <package> depends — **[Results shown in Figure 4(i).]**
  —I specify an exact version number
  —I specify a range of version numbers, e.g., 3.x.x, or [2.1 through 2.4]
  —I specify just a package name and always get the newest version
  —I specify a range or just the name, but I take a snapshot of dependencies (e.g., shrinkwrap, packrat)
- What is the common practice in <ecosystem> for declaring version numbers of dependencies? — [same scale as previous + "don't know"]

Using or avoiding dependencies.

- When adding a dependency to <package>, I usually do significant research to assess the quality of the package or its maintainers, before relying on a package that seems to provide the functionality I need. — [agreement scale, as above]**[Results shown in Figure 4(d).]**
- It's only worth adding a dependency if it adds a substantial amount of value. — [agreement scale, as above]**[Results shown in Figure 4(c).]**
- I often choose NOT to update <package> to use the latest version of its dependencies. — [agreement scale, as above]**[Results shown in Figure 4(h).]**
- When adding a dependency, I usually create an abstraction layer (i.e., facade, wrapper, shim) to protect internals of my code from changes. — [agreement scale, as above]
- When working on <package>, I often copy or rewrite segments of code from other packages into my package, to avoid creating a new dependency. — [agreement scale, as above]
- When working on <package>, I must expend substantial effort to find versions of all my dependencies that will work together. — [agreement scale, as above]
- (OPTIONAL) Compare <ecosystem> with other ecosystems you've used or heard about – does one have some features that the other should adopt? If so, name the other ecosystem(s) and describe the feature(s). — [text field]
- (OPTIONAL) Why do you think people chose to design these other ecosystem(s) differently from <ecosystem>? — [text field]

Part V: Demographics and motivations.

- Age
  —18–24
  —25–34

  — 35–44
  — 45–54
  — 55–64
  — 65+
- Gender — [male/female/other]
- Formal computer science education/training
  — None
  — Coursework
  — Degree
- How many years have you been contributing to open source? (in any way, including writing code, documentation, engaging in discussions, etc) — [same time scale as "years used ecosystem" above]
- How many years have you been developing or maintaining software? — [same as previous]
- (OPTIONAL) Is there anything else we should have asked, that would help us better understand your experience with community values and breaking changes in <ecosystem> If so, tell us about it: — [text field]

C SUGGESTED SET OF VALUES FOR FUTURE STUDIES

We propose the following list of values that appear to distinguish software ecosystems. They are derived from Study 1 results plus examination of ecosystem webpages, then modified based on survey results, adding values that were suggested by survey respondents (Standardization, Technical Diversity, Usability, and Social Benevolence), and removing one that does not distinguish meaningfully among developers or ecosystems (Quality).

- *Stability:* Backward compatibility, allowing seamless updates ("do not break existing clients").
- *Innovation:* Innovation through fast and potentially disruptive changes.
- *Replicability:* Long-term archival of current and historic versions with guaranteed integrity, such that exact behavior of code can be replicated.
- *Compatibility:* Protecting downstream developers and end-users from struggling to find a compatible set of versions of different packages.
- *Rapid Access:* Getting package changes through to end-users quickly after their release ("no delays").
- *Commerce:* Helping professionals build commercial software.
- *Community:* Collaboration and communication among developers.
- *Openness* and Fairness: ensuring that everyone in the community has a say in decision-making and the community's direction.
- *Curation:* Selecting a set of consistent, compatible packages that cover users' needs.
- *Fun and personal growth:* Providing a good experience for package developers and users.
- *Standardization:* Promote standard tools and practices, limiting developers choice to save them time and effort.
- *Technical Diversity:* Allowing developers freedom to develop and interact in a diversity of ways.
- *Usability:* Ensuring that tools and libraries are easy for developers to use; ensuring resulting software is easy for end-users to use.
- *Social Benevolence:* An ethical community empowering others by making software and other resources available.

D LOCK FILE NAMES IN EACH ECOSYSTEM

Table 11. Lock Files Counted in Different Ecosystems

Ecosystem	Lock file	Notes
Atom (plugins)	package-lock.json, npm-shinkwrap.json	(see Node.js/NPM below)
CocoaPods	podfile.lock	
Eclipse (plugins)	N/A	This function would be done within the project's regular metadata files (plugin.xml and pom.xml) and so could not be measured readily with this technique
Erlang,Elixir/Hex	mix.lock	
Go	GoPkg.lock, vendor/	Preceding the GoPkg.lock file, a canonical method of locking down dependency versions was to simply include a snapshot of their source code; so we looked for a "vendor/" directory in the project.
Haskell (Cabal/Hackage)	cabal.config	
Haskell (Stack/Stackage)	cabal.config	Although possible, this was never used since Stackage's main distinguishing feature is to constrain the versions of a set of packages
Lua/Luarocks	N/A	We could not find evidence of a canonical or even common practice way of locking down Lua versions
Maven	N/A	This function would be done within the project's regular metadata file (pom.xml) and so could not be measured readily with this technique
Node.js/NPM	package-lock.json, npm-shinkwrap.json	These are both npm lockfiles with some semantic differences;[27] npm-shrinkwrap is intended to be published; package-lock is not; however, both can be found in GitHub projects.
NuGet	project.lock.json	The NuGet blog suggests saving this file to a repository to lock in dependency versions.[28]
Perl/CPAN	cpanfile.snapshot	We could not find evidence of a canonical way to do this in CPAN, but one recommendation was a third-party package called *Carton*[29] that creates this snapshot file.
PHP/Packagist	composer.lock	
Python/PyPi	N/A	We could not find evidence of a canonical way to do this in Pypi; a StackOverflow post suggested that there are several nonstandard alternatives.[30]
R/Bioconductor	packrat.lock	Not canonically standard, but common and well-known. However, it is mostly irrelevant for Bioconductor, since a set of mutually compatible packages are released as a unit.
R/CRAN	packrat.lock	Not canonically standard, but common and well-known.
Ruby/Rubygems	Gemfile.lock	
Rust/Cargo	Cargo.lock	

[27] https://docs.npmjs.com/files/package-lock.json.
[28] https://blog.nuget.org/20181217/Enable-repeatable-package-restores-using-a-lock-file.html.
[29] https://metacpan.org/pod/Carton.
[30] https://stackoverflow.com/questions/8726207/what-are-the-python-equivalents-to-rubys-bundler-perls-carton.

ACKNOWLEDGMENTS

We want to thank Audris Mockus and the WoC project at University of Tennessee, Knoxville, for access to the WoC archive [57] for data mining, and the many people interviewed and surveyed, and those who helped with the design and promotion of the survey.

REFERENCES

[1] Pietro Abate, Roberto DiCosmo, Ralf Treinen, and Stefano Zacchiroli. 2011. MPM: A modular package manager. In *Proceedings of the International Symposium on Component Based Software Engineering (CBSE'11)*. ACM Press, New York, 179–188. DOI: https://doi.org/10.1145/2000229.2000255

[2] Rabe Abdalkareem. 2017. Reasons and drawbacks of using trivial npm packages: The developers' perspective. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, New York, NY, 1062–1064.

[3] Cyrille Artho, Kuniyasu Suzaki, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. 2012. Why do software packages conflict? IEEE International Working Conference on Mining Software Repositories, 141–150.

[4] Anat Bardi and Shalom H. Schwartz. 2003. Values and behavior: Strength and structure of relations. *Personal. Soc. Psychol. Bull.* 29, 10 (2003), 1207–1220.

[5] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache community upgrades dependencies: An evolutionary study. *Empir. Softw. Eng.* 20, 5 (2015), 1275–1317.

[6] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: Cost negotiation and community values in three software ecosystems. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE'16)*. ACM Press, New York.

[7] Christopher Bogart, Anna Filippova, James Herbsleb, and Christian Kastner. 2017. Culture and Breaking Change: A Survey of Values and Practices in 18 Open Source Software Ecosystems. DOI: https://doi.org/10.1184/R1/5108716.v1

[8] Shawn A. Bohner and Robert S. Arnold. 1996. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA.

[9] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualit. Res. Psychol.* 3, 2 (2006), 77–101. DOI: https://doi.org/10.1191/1478088706qp063oa

[10] A. Brito, L. Xavier, A. Hora, and M. T. Valente. 2018. Why and how Java developers break APIs. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*. 255–265.

[11] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2015. Enabling the definition and enforcement of governance rules in open source systems. *Proceedings of the International Conference on Software Engineering (ICSE'15)*. 505–514. DOI: https://doi.org/10.1109/ICSE.2015.184

[12] Jaepil Choi and Heli Wang. 2007. The promise of a managerial values approach to corporate philanthropy. *J. Bus. Ethics* 75, 4 (2007), 345–359.

[13] Juliet Corbin and Anselm Strauss. 2014. Criteria for evaluation. In *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory* (3rd ed.). Sage Publications, Inc.

[14] Bradley E. Cossette and Robert J. Walker. 2012. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE'12)*. ACM Press, New York, 55.

[15] John W. Creswell and J. David Creswell. 2014. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches* (4th ed.). Sage Publications.

[16] Mary Crossan, Daina Mazutis, and Gerard Seijts. 2013. In search of virtue: The role of virtues, values and character strengths in ethical decision making. *J. Bus. Ethics* 113, 4 (2013), 567–581.

[17] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: Transparency and collaboration in an open software repository. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'12)*. 1277–1286.

[18] Barthélémy Dagenais and Martin P. Robillard. 2010. Creating and evolving developer documentation: Understanding the decisions of open source contributors. In *Proceedings of the ACM International Symposium on Foundations of Software Engineering*. 127–136. DOI: https://doi.org/10.1145/1882291.1882312

[19] Cleidson R. B. de Souza and David F. Redmiles. 2008. An empirical study of software developers' management of dependencies and changes. In *Proceedings of the International Conference on Software Engineering (ICSE'08)*.

[20] Cleidson R. B. De Souza and David F. Redmiles. 2009. On the roles of APIs in the coordination of collaborative software development. *Comput. Supp. Coop. Work* 18, 5-6 (2009), 445–475. DOI: https://doi.org/10.1007/s10606-009-9101-3

[21] Alexandre Decan, Tom Mens, Maëlick Claes, and Philippe Grosjean. 2016. When GitHub meets CRAN: An analysis of inter-repository package dependency problems. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. 493–504. DOI: https://doi.org/10.1109/SANER.2016.12

[22] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER'17)*.

[23] Dedoose. 2016. Version 7.0.23. *Web Application for Managing, Analyzing, and Presenting Qualitative and Mixed Method Research Data. SocioCultural Research Consultants, LLC, Los Angeles, CA*. Retrieved from www.dedoose.com

[24] Jim des Rivières. 2005. API First. Retrieved from http://www.eclipsecon.org/2005/presentations/EclipseCon2005_12.2APIFirst.pdf.

[25] Jim des Rivières. 2007. Evolving Java-based APIs. Retrieved from https://wiki.eclipse.org/Evolving_Java-based_APIs.

[26] Jens Dietrich, David J. Pearce, Jacob Stringer, and Kelly Blincoe. 2019. Dependency versioning in the wild. In *Proceedings of the Conference on Mining Software Repositories (MSR'19)*. 349–359. DOI: https://doi.org/10.1109/MSR.2019.00061

[27] Don A. Dillman, Jolene D. Smyth, and Leah Melani Christian. 2014. *Internet, Phone, Mail, and Mixed-mode Surveys: The Tailored Design Method*. John Wiley & Sons.

[28] Alexander Eck. 2018. Coordination across open source software communities: Findings from the rails ecosystem. In *Tagungsband Multikonferenz Wirtschaftsinformatik (MKWI'18)*. 109–120.

[29] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. 2001. Does code decay? Assessing the evidence from change management data. *IEEE Trans. Softw. Eng.* 27, 1 (Jan. 2001), 1–12. DOI: https://doi.org/10.1109/32.895984

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA.

[31] R. Stuart Geiger. 2017. Summary analysis of the 2017 GitHub open source survey. *CoRR* abs/1706.02777 (2017).

[32] Gemnasium. 2017. Gemnasium. Retrieved on 28 April, 2021 from https://web.archive.org/web/20180324121439/https://gemnasium.com/.

[33] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. 2017. Some from here, some from there: Cross-project code reuse in GitHub. In *Proceedings of the IEEE International Working Conference on Mining Software Repositories*. 291–301. DOI: https://doi.org/10.1109/MSR.2017.15

[34] GitHub, Inc. 2017. Open Source Survey 2017. Retrieved from http://opensourcesurvey.org/2017/ on 4/28/2021.

[35] The Neighbourhoodie Software GmbH. 2017. Greenkeeper.io. Retrieved on 28 April, 2021 from https://web.archive.org/web/20180224075015/https://greenkeeper.io/.

[36] Johannes Henkel and Amer Diwan. 2005. CatchUp!: Capturing and replaying refactorings to support API evolution. In *Proceedings of the International Conference on Software Engineering (ICSE'05)*. ACM Press, New York, 274–283.

[37] Steven Hitlin and Jane Allyn Piliavin. 2004. Values: Reviving a dormant concept. *Ann. Rev. Sociol.* 30, 1 (2004), 359–393.

[38] Reid Holmes and Robert J. Walker. 2010. Customized awareness: Recommending relevant external change events. In *Proceedings of the International Conference on Software Engineering (ICSE'10)*. ACM Press, New York, 465–474. DOI: https://doi.org/10.1145/1806799.1806867

[39] Daqing Hou and Xiaojia Yao. 2011. Exploring the intent behind API evolution: A case study. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'11)*. IEEE Computer Society, Los Alamitos, CA, 131–140.

[40] Marco Iansiti and Roy Levien. 2004. *The Keystone Advantage: What the New Dynamics of Business Ecosystems Mean for Strategy, Innovation, and Sustainability*. Harvard Business Press, Boston, MA.

[41] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2015. Enabling the definition and enforcement of governance rules in open source systems. In *Proceedings of the International Conference on Software Engineering (ICSE'15)*. IEEE, 505–514.

[42] Steven J. Jackson, David Ribes, Ayse G. Buyuktur, and Geoffrey C. Bowker. 2011. Collaborative rhythm: Temporal dissonance and alignment in collaborative scientific work. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'11)*. 245–254.

[43] Slinger Jansen and Michael A. Cusumano. 2013. Defining software ecosystems: A survey of software platforms and business network governance. In *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar Publishing.

[44] Puneet Kapur, Brad Cossette, and Robert J. Walker. 2010. Refactoring references for library migration. In *Proceedings of the International Conference on Object-oriented Programming, Systems, Languages and Applications (OOPSLA'10)*. ACM Press, New York, 726–738. DOI: https://doi.org/10.1145/1869459.1869518

[45] Smitha Keertipati, Sherlock A. Licorish, and Bastin Tony Roy Savarimuthu. 2016. Exploring decision-making processes in Python. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*. ACM, 43.

[46] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR'17)*. IEEE Press, Piscataway, NJ, 102–112.

[47] Daniel Le Berre and Pascal Rapicault. 2009. Dependency management for the eclipse ecosystem: Eclipse P2, metadata and resolution. In *Proceedings of the International Workshop on Open Component Ecosystems (IWOCE'09)*. 21–30. DOI: https://doi.org/10.1145/1595800.1595805

[48] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API change and fault proneness: A threat to the success of Android apps. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE'13)*. ACM Press, New York, 477–487.

[49] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: A map of code duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 1–28. DOI: https://doi.org/10.1145/3133908

[50] Mircea F. Lungu. 2009. *Reverse Engineering Software Ecosystems*. Ph.D. Dissertation. University of Lugano.

[51] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. 2006. Managing the complexity of large free and open source package-based software distributions. 199–208. DOI: https://doi.org/10.1109/ASE.2006.49

[52] Konstantinos Manikas. 2016. Revisiting software ecosystems research: A longitudinal literature study. *J. Syst. Softw.* 117 (2016), 84–103.

[53] Michael Mattsson and Jan Bosch. 2000. Stability assessment of evolving industrial object-oriented frameworks. *J. Softw. Maint.: Res. Pract.* 12, 2 (2000), 79–102.

[54] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of API stability and adoption in the Android ecosystem. In *Proceedings of the International Conference on Software Maintenance (ICSM'13)*. IEEE Computer Society, Los Alamitos, CA.

[55] T. Mens. 2016. An ecosystemic and socio-technical view on software maintenance and evolution. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'16)*. 1–8.

[56] David G. Messerschmitt, Clemens Szyperski et al. 2005. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press Books.

[57] Audris Mockus. 2009. Amassing and indexing a large sample of version control systems: Towards the census of public soruce code history. In *Proceedings of the IEEE Conference on Mining Software Repositories (MSR'09)*.

[58] Emerson Murphy-Hill, Thomas Zimmerman, and Nachiappan Nagappan. 2014. Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In *Proceedings of the International Conference on Software Engineering (ICSE'14)*. DOI: https://doi.org/10.1145/2568225.2568226

[59] Linda Northrop, Peter Feiler, Richard P. Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Douglas Schmidt, Kevin Sullivan, and Kurt Wallnau. 2006. *Ultra-large-scale Systems: The Software Challenge of the Future*. Software Engineering Institute.

[60] Siobhán O'Mahony and Fabrizio Ferraro. 2007. The emergence of governance in an open source community. *Acad. Manag. J.* 50, 5 (2007), 1079–1106.

[61] Jeroen Ooms. 2013. Possible directions for improving dependency versioning in R. *R Journal* 5, 1 (2013), 1–9.

[62] Klaus Ostermann, Paolo G. Giarrusso, Christian Kästner, and Tillmann Rendel. 2011. Revisiting information hiding: Reflections on classical and nonclassical modularity. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'11) (Lecture Notes in Computer Science)*, Vol. 6813. Springer-Verlag, Berlin, 155–178.

[63] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (1972), 1053–1058. DOI: https://doi.org/10.1145/361598.361623

[64] Raphael Pham, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider. 2013. Creating a shared understanding of testing culture on a social coding site. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. IEEE Computer Society, Los Alamitos, CA, 112–121.

[65] Tom Preston-Werner. 2013. Semantic Versioning 2.0.0. Retrieved from http://semver.org.

[66] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2012. Measuring software library stability through historical version analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM'12)*. IEEE Computer Society, Los Alamitos, CA, 378–387.

[67] Steven Raemaekers, Arie Van Deursen, and Joost Visser. 2014. Semantic versioning versus breaking changes: A study of the Maven repository. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*. IEEE Computer Society, Los Alamitos, CA, 215–224. DOI: https://doi.org/10.1109/SCAM.2014.30

[68] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How do developers react to API deprecation? The case of a smalltalk ecosystem. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*. ACM Press, New York. DOI: https://doi.org/10.1145/2393596.2393662

[69] RStudio Team. 2015. *RStudio: Integrated Development for R*.Technical Report. RStudio, Inc., Boston MA. Retrieved from www.rstudio.com

[70] Edgar H. Schein and Peter Schein. 2017. *Organizational Culture and Leadership* (5th ed.). Wiley.
[71] Shalom H. Schwartz. 1992. Universals in the content and structure of values: Theoretical advances and empirical tests in 20 countries. *Adv. Exper. Soc. Psychol.* 25 (1992), 1–65.
[72] Leif Singer, Fernando Figueira Filho, and Margaret-Anne Storey. 2014. Software engineering at the speed of light: How developers stay current using Twitter. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*. 211–221. DOI: https://doi.org/10.1145/2568225.2568305
[73] Ian Sommerville. 2010. *Software Engineering* (9th ed.). Pearson Addison Wesley.
[74] Diomidis Spinellis. 2012. Package management systems. *IEEE Softw.* 29, 2 (2012), 84–86.
[75] Adam Stakoviak, Andrew Thorp, and Isaac Schleuter. 2013. The Changelog. Retrieved from https://changelog.com/101/.
[76] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr.1999. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering (ICSE'99)*. IEEE Computer Society, Los Alamitos, CA, 107–119.
[77] The LibreOffice Design Team. 2017. What Open Source Means To LibreOffice Users. Retrieved from https://design.blog.documentfoundation.org/2017/09/13/open-source-means-libreoffice-users/.
[78] The Rust Team. 2021. The Cargo Book. Retrieved on 28 April, 2021 from https://doc.rust-lang.org/cargo/faq.html#why-do-binaries-have-cargolock-in-version-control-but-not-libraries.
[79] Jonathan Tuner. 2016. State of Rust Survey 2016. Retrieved from https://blog.rust-lang.org/2016/06/30/State-of-Rust-Survey-2016.html.
[80] A. Turon and N. Matsakis. 2014. Stability as a Deliverable (The Rust Programming Language Blog). Retrieved from https://blog.rust-lang.org/2014/10/30/Stability.html.
[81] Ivo van den Berk, Slinger Jansen, and Lútzen Luinenburg. 2010. Software ecosystems. In *Proceedings of the European Conference on Software Architecture (ECSA'10)*. 127–134. DOI: https://doi.org/10.1145/1842752.1842781
[82] Bill Venners. 2003. The Philosophy of Ruby: A Conversation with Yukihiro Matsumoto, Part I. Retrieved from http://www.artima.com/intv/rubyP.html.
[83] Jonathan Wareham, Paul B. Fox, and Josep Lluís Cano Giner. 2014. Technology ecosystem governance. *Organiz. Sci.* 25, 4 (2014), 1195–1215.
[84] Mark Weiser. 1984. Program slicing.*IEEE Trans. Softw. Eng.* 10, 4 (1984), 352–357.
[85] Joel West. 2003. How open is open enough?: Melding proprietary and open source platform strategies. *Res. Polic.* 32, 7 (2003), 1259–1285.
[86] Joel West and Siobhán O'Mahony. 2008. The role of participation architecture in growing sponsored open source communities. *Industr. Innov.* 15, 2 (2008), 145–168.
[87] Hadley Wickham. 2015. *Releasing a Package*. O'Reilly Media, Sebastopol, CA. Retrieved from http://r-pkgs.had.co.nz/release.html.
[88] Wei Wu, Foutse Khomh, Bram Adams, Yann Gaël Guéhéneuc, and Giuliano Antoniol. 2015. An exploratory study of API changes and usages based on Apache and Eclipse ecosystems. *Empir. Softw. Eng.* (2015), 1–47. DOI: https://doi.org/10.1007/s10664-015-9411-7
[89] Wei Wu, Foutse Khomh, Bram Adams, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2016. An exploratory study of API changes and usages based on Apache and Eclipse ecosystems. *Empir. Softw. Eng.* 21, 6 (2016), 2366–2412.
[90] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*. IEEE, 138–147.
[91] Yihui Xie. 2013. R Package Versioning. Retrieved from http://yihui.name/en/2013/06/r-package-versioning/.
[92] Robert A. Yin. 2013. *Case Study Research: Design and Methods* (5th ed.). Sage Publications.

Received August 2019; revised December 2020; accepted January 2021