Open Source Projects: Best Practices for VisionEval

Dan Flynn
Volpe Center
daniel.flynn.ctr@dot.gov

Summary

This document reviews proposed Best Practices in Collaborative Open Source Software Development, to support the VisionEval application framework and transportation planning models.

Open source development provides a platform for users and developers to engage in continuous, collaborative improvement of software tools. Collaboration via open source software development aligns with goals of public agencies involved in transportation planning, who need to support freely-available decisionmaking tools which can be continuously improved by the user community and widely disseminated.

VisionEval is an open source framework for building disaggregate strategic planning models for transportation planners. These models are currently oriented towards providing easy-to use tools to visualize the impact of potential transportation policies, with target users being planners at state agencies and metropolitan planning organizations. The suite of modeling tools in the VisionEval framework currently includes GreenSTEP, the Regional Strategic Planning Model (RSPM), the Rapid Policy Assessment Tool (RPAT), and the Energy and Emissions Reduction Policy Analysis Tool (EERPAT). The latter three build on GreenSTEP, but emphasize different geographic scales (regional to State-level) and planning topics (land use, energy use, and greenhouse gas emissions).

This open-source framework is the result of collaborations among several partner agencies, including the Federal Highway Administration (FHWA), the Oregon Department of Transportation (OregonDOT), and the American Association of State Highway and Transportation Officials (AASHTO). A transportation pooled fund supported by a number of States and Metropolitan Planning Organizations (MPOs) was established in 2017 to continue VisionEval development and application.

Because the VisionEval framework supports a broad array of potential applications that will be developed by a community that is likely to grow beyond the current partners, the pooled fund will develop a plan for incorporating contributed software that will extend the current functionality of the framework and that will add additional modules and models. This paper provides an overview of best practices for collaborative open source software development. In particular, this paper outlines the process by which contributions will be evaluated and reviewed, and describes how version control and collaboration tools can be used to facilitate that process.

Table of Contents

1. Introduction

Many open source software projects are developed by Internet-based communities of software developers, who voluntarily collaborate their code to build a software tool for the public benefit (Von Krogh 2003). Open

source projects may attract a user and development community which spurs innovation and novel applications beyond what the original developers may have envisioned (Bettenburg et al. 2015), and they have become core parts of numerous commercial and non-commercial software projects. In the public sector, open source approaches have become the common in agencies as diverse as the Department of Defense, NASA, and the Consumer Financial Protection Bureau.

Governmental users and organizations of open source collaboration platforms such as GitHub have been growing exponentially. GitHub has established a site to facilitate application of open source principles to governmental software development (government.github.com). For public agencies, major benefits of adopting these approaches to software development include the transparency, cost-effectiveness, and longevity of open source tools.

Transparency in open source development is achieved because code is readily available to the community of users. This promotes accountability and reproducibility of research projects, beyond what can be achieved even by peer-review of a journal article. Some researchers argue that the element of transparency also promotes innovation, as modules of code can be reviewed and built on, spurring rapid development of tools in as areas such as information systems (Von Krogh and Spaeth 2007) and weather station monitoring (Heistermann et al. 2015).

Cost-effectiveness comes from the ability to build new applications using shared tools, reducing the amount of new code development required. For example, the software QGIS is an open source geospatial modeling and visualization tool, and builds on a large number of modules written in the programming language Python. Users can add modules within QGIS for their own specific needs, and an active community of users discusses these suggested modifications and decides how to incorporate them into the project. Using the resulting software tools is free and the tools themselves are constantly improving. Resources can be focused on addressing immediate needs, with much less expense building the foundation for the overall software. The free availability of this spatial analysis tool has made QGIS popular, for instance, with water resource managers in developing countries (Chen et al. 2010).

Longevity arises from the fact that the code is developed in a community effort. As understanding of the code base is not tied to a single individual, the code can continue to be maintained and supported by the community. Provided the community of knowledgeable developers exceeds a certain size, the project can be resilient to changes in the involvement of individual developers (Heistermann et al. 2015). For public agencies, the community of developers avoids lock-in with a single vendor (Zhu and Zhou 2012), and allows the same code base to be enhanced through successive contracts with many potential vendors.

A crucial factor underpinning these benefits is the licensing that protects open source software. The 'openness' of open source projects does not mean that they have no ownership or license; open source software is often protected by copyrights and offered to the public through licenses such as the General Public License (GPL). The original GPL license, developed in the 1980's by the Free Software Foundation (Von Krogh 2003) ensured that the right to use, study, modify, and distribute modified or unmodified code at no cost. Licences are key to preserving open source development, allowing users to contribute code without concerns that their contributions will cease to be available in the future.

Software licenses can be chosen to reflect the needs of the project, using tools such as https://choosealicense.com/. The key elements of an open source license include the the following:

- Permission to modify code
- Permission to use commercially or privately
- Permission to distribute freely
- Attribution of authorship
- Limitations of liability and warranty
- Ability to patent the elements of code contributed
- Requirement that any derived code follow the license of the original code

The first three elements are common to all open source licenses. Attribution of authorship is typical, and ensures that derivative work based on the original code acknowledge the earlier author(s). For example, the Apache 2.0 license states:

> You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works.

Patent protection is granted by some licenses, such as the Apache 2.0 license. This means that commercial ventures may contribute code to an open source project and still apply for a patent on that contributed code.

For a research project such as VisionEval, a permissive license such as the MIT license or Apache 2.0 license would balance the need for preservation of copyright and attribution of authorship while promoting widespread use. The former is more permissive, while the latter adds grant of patent rights, and adds the requirement that changes to the code must be stated. A more restrictive license, such as the GPL, requires that derivative code be distributed under the terms of the original license. That restriction makes it more difficult for proprietary software to include the open source code, but conversely, it allows the original developer to charge money to license the same code under different terms for proprietary uses, as has been done by MySQL, the Qt graphical interface, and various software products released by Oracle. Collaborative projects, such as the R Project for Statistical Computing, may use a more restrictive license to ensure that private developers do not profit from the project without making contributions back to the original effort.

An additional layer of licensing can be used to ensure that contributions to a project fall within the license of the project. Contributor License Agreements (CLAs) ensure that when a project accepts contributions from a third party, all necessary copyrights to release the code are in place. A CLA can provide extra assurance that the governing organization of a project has all the appropriate permissions to incorporate patches or improvements to the project, and distribute the resulting work. A CLA may be unnecessary for open source project where concerns about patent or copyright infringement are minimal, and is mostly useful when a for-profit company submits patented code to an open-source project. For a project such as VisionEval, a CLA likely is not essential.

2. Open source governance

Achieving these benefits of open source tools requires a governance structure. There are a variety of governance models available for open source projects. The distinction between commercial software projects and non-profit software projects provides a useful baseline for evaluating governance mechanism. The Android and Linux operating systems, for example, are both open source, and share an overall similar structure of how code is reviewed, with some crucial differences reflecting the fact that Linux is managed via an open community, whereas Android is a Google product (Bettenburg et al. 2015).

Common to both approaches, suggestions are reviewed by the community to assure quality, determine fit of the contribution to the project, and to "sanitize" code (ensuring that it does not present security risks). If the code meets the review criteria, it will be accepted into the main project repository. This process is explained in detail in the Open Source Development section, and is outlined in the figure below.

For both commercial and non-commercial open source projects, this same work flow is followed, with the distinction being the identity of the contributors and degree to which outside contributions are accepted into the code base.

The commercial Android software is developed by Google and distributed freely, with the core software developed by programmers supported by Google. Phone hardware manufactures can modify the Android software to tailor it to their devices, and users can submit modifications to be integrated into the core software, acting as Contributors in the process above. Contributors are required to agree to a Contributor License Agreement (CLA, see above) to transfer intellectual property rights to Google. This is a centralized project, with staff supported by Google to review code, who act to implement the Review, Verification, and Integration steps.

For the non-commercial Linux operating system, no CLA is required for contributors. This large, decentralized project has many contributors and many reviewers, without a single review team. The difference between centralized and decentralized projects has been likened to the difference between a 'cathedral' and a 'bazaar' (Raymond 1999). The 'bazaar' style of Linux means that new ideas are constantly being shared and reviewed,

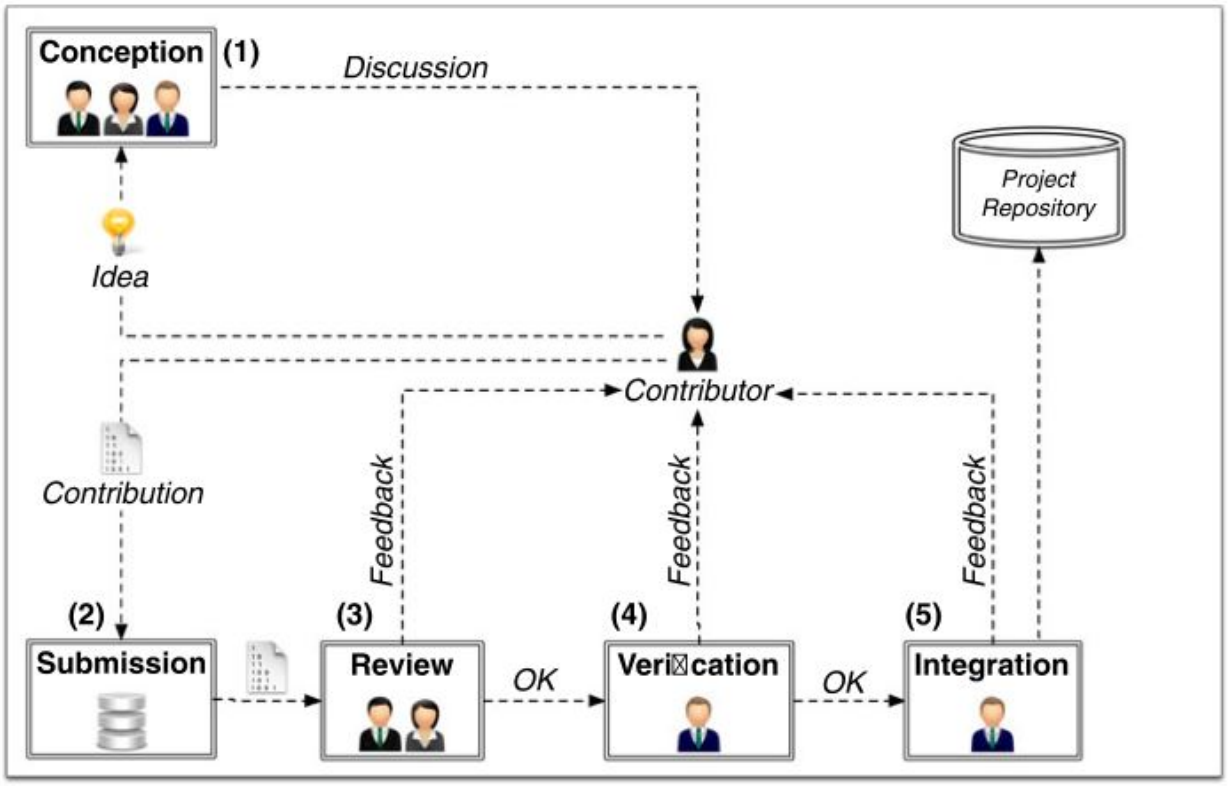

Figure 1: Figure 1. Conceptual model of the collaboration management process (Bettenburg et al. 2015).

far more than any one team could manage. Commercial software and smaller teams typically take more of a 'cathedral' style development process, where only a core group of developers has the authority to commit changes to the central repository. Some projects, such the R statistical software, exhibit both approaches: the core R software is managed like a 'cathedral', whereas R extension packages are presented in various 'bazaars' operated by the R project itself and by outside organizations such as BioConductor, OmegaHat, or GitHub.

Further examples of governance structures for code review follow below. These examples highlight the differences between large projects where many modules are being developed in an experimental fashion, with bazaar-style exchange of ideas and multiple versions, versus smaller projects curated in a cathedral-style fashion. For VisionEval, maintaining a balance between 'cathedral' and 'bazaar' style development will be essential to encourage users to be active in experimenting with new modules, while maintaining a functional and stable code base.

Example large project: `Node.js`

For a large open source project, extensive community engagement and a clear governance structure is essential. Node.js is a JavaScript tool for event-driven web programming. The community is organized into groups of Users, Collaborators, and Leaders. The number of users is vast, at least in the tens of thousands. At the leadership level, there is a Core Technical Committee (CTC) of approximately 20 members. Additional Working Groups exist for to manage security, specific technical issues, and releases.

Users can most easily become involved in this coding effort by contributing issues on the project GitHub page, where over 5,000 issues have been submitted. If users decide to contribute a new feature or propose a bug fix, they can open a "pull request" to ask code owners to consider their contribution. Over 8,000 pull requests have been opened for Node.js, with some receiving comments from over 100 users or Collaborators. An extensive set of development guidelines exists. These guidelines specify, among other issues, that contributors state a "Certificate of Origin" which acts similarly to a CLA to ensure that contributions will remain public and consistent with the license of the project itself.

Users become Collaborators by being identified by the CTC as someone who has made signfificant and valuable contributions to the project. Collaborators review contributions of code, help other collaborators, and have the right to accept pull requests into the code repository. The Core Technical Committee has oversight of the activities of the collaborators.

New CTC members are considered and accepted by existing CTC members, at one of the weekly meetings that occurs. A unique feature of the Node.js governance is that no more than 1/3 of the CTC members may be affiliated with the same employer. Decisions are made by "Consensus Seeking" process, whereby if the CTC cannot come to consensus on an issue, it can either be tabled or put forward for a vote by simple majority.

Community engagement is extensive, with a web chat channel, Twitter account, weekly newsletter, podcast, and many web resources. User groups exist in many languages, and documentation exists in many forms. These community engagement efforts are key to helping new users learn about the tool and become proficient. While VisionEval will not likely reach the same scale of adoption, the clear structure used by this mature open source project provides a model to examine while considering a governance structure.

Example small project: `vegan`

`vegan` is a popular package of R tools for ecologists, with a focus on analysis of vegetation data. There are on average 800 weekly downloads of the package, and thousands of users. The package was developed originally by a single academic researcher, and now has 16 active contributors, and has been forked over 30 times. With vastly smaller scale than Node.js, this community of developers has not developed any formal governance structure, apparently relying on informal email communication between developers to coordinate pull requests.

Some of the documentation of `vegan` is maintained by the single original developer, with the package itself containing documentation vetted by the contributors. The package is licensed under GPL-2, a fairly permissive

copyleft license.[1] This license or a similar one is recommended for scientific research projects (Wilson 2017), allowing reuse in commercial or noncommercial settings while guaranteeing access to the original source code. Other than the license, this project lacks any formal governance structure, which may be common for open source projects which were designed by single users, even with a relatively large base of users.

While this informal structure clearly works for a tool with an active academic group of users, a clear governance structure is likely to benefit a project such as VisionEval. For VisionEval, the mix of public agency, consultant, and academic users is both larger and more heterogeneous than the small academic project, so clear definition of roles and responsibilities of users, contributors, and leadership will be beneficial.

Example travel model project: `ActivitySim`

ActivitySim provides a platform for activity-based travel modeling. Like Node.js, ActivitySim has a clearly defined governance structure and development workflow. ActivitySim uses GitHub (see below) to manage the workflow, with a stable version hosted on the master branch, and improvements submitted via pull requests. The governance structure differs from Node.js and similar initiatives because the effort is a pooled effort by a small number of organizations working through the Association of Municipal Planning Organizations (AMPO). Thus the governance structure is as follows:

- **Users** of the modeling tool may submit issues through GitHub to notify developers about bugs or request features.
- **Developers** contribute code or documentation, and act in the "Contributor" role in Figure 1 above.
- **Committers** can accept code from developers, and have signed a CLA. Committers act in all three of the Review, Verification, and Integration steps in the figure. Committers are designated by the Project Management Committee.
- **Funders** support the development of the code and project administration, and may or may not be involved in the day-to-day of the project.
- **Project Mangagement Committee** (PMC) members serve multiple roles. The PMC is composed of funders initially, but could include developers or committers in the future. As a group, the PMC provides oversight to the whole project. There is a PMC contractor, who acts as the lead organization to carry out the software development and administration, and is the most active component of the Developer and Committer groups.

ActivitySim has a detailed decision-making process, where the most common action (code change) requires no voting, with implicit approval unless a PMC member disagrees. This is also known as "lazy approval" or "lazy consensus", and has been used often in other open source projects (e.g., Apache). Another open source project (`dash.js`) offer reasons why such implicit consensus is preferable for technical decisions:

> The vast majority of decisions in an open source project are technical in nature and can therefore be driven by direct contributions to testing, feature management, documentation and code. All such decisions are cheap and easily reversible. We therefore make such decisions by "Lazy Consensus" - a process in which those with merit are trusted to "do the right thing". Non-technical decisions are often harder to reverse and are therefore made through "active consensus" - a process in which anyone can actively build consensus amongst those with merit.

ActivitySim provides a useful model for VisionEval. Important decisions will include what the composition of the PMC or similar oversight group will be, what decision-making framework the PMC will follow, and how Committers and Developers will be designated. The remainder of this document illustrates how VisionEval development might be managed through such a structure.

[1] "Copyleft" is a term coined by the GNU project to refer to licenses that protect the rights of future users of the software to copy, study and modify the original code as well as code derived from it. More permissive licenses such as the Apache license allow subsequent developers to assert proprietary rights over derivative works.

3. Open source development

Open source development relies on a community of developers to produce, evaluate, and integrate code into a project. There are a number ways that development can be organized in an open source framework, but the most important features are ability to incorporate existing code from other projects, ability to develop features independently, and agreed-upon methods for developers to share their additions with the community of other developers and users.

Incorporating code from other other projects is central to open source development. For example, the Vision California Urban Footprint project is an open-source scenario development and analysis tool for land use planning (Hosted on GitHub). This project drew from 12 distinct open source projects, including tools for database management, geospatial analysis, and data processing. The open source framework of this project has encouraged multiple users to adopt and modify the tool.

Workflow for VisionEval

The VisionEval code and documentation are currently maintained on GitHub. Consequently, it will be simplest to follow the typical `git` work flow, using the GitHub web interface as described in detail below. In general, the `git` workflow involves creation of a repository (folder of files) of the working version of the code, which is distributed to all the developers. The distributed nature of the repository means that developers can independently make modifications and additions to the code, and then request that this code be incorporated into the main version to be shared with all other users. The details of this work flow are outlined here as a starting point.

The initial repository establishment might start by establishing the VisionEval framework itself. This repository initialization can be done in several ways, one of which is outlined in the Appendix. This procedure, which transfers repository ownership, would add code to the VisionEval project. This would be the version 1.0 release (or other version number), and would become the new master branch. Following the typical GitHub workflow (see here and below), developers would then add code as necessary in their own development branches to resolve bugs and extend the model with additional features. Contributions from the community of developers would be reviewed and accepted if useful into the master branch, see below. Documentation of the software tool, essential to grow the community of users, would then be completed, and a new stable version would be released after passing integration tests using tools such as Travis-CI.

The workflow for VisionEval, including the structure of the repository, will be determined by the Pooled Fund members. In general, the VisionEval repository will house the master branch of each of the strategic planning tools, and may also house a testing branch used to evaluate incoming contributions. The methods of reviewing code contributions will need to be established by the pooled fund members; an overview of code review procedures follows below.

Reviewing contributions from multiple sources

Unlike proprietary software projects, open source research projects frequently distinguish between two public communities: developers and users. Contributions from the user community may be rare, typically limited to testing and bug reporting (Aksulu and Wade 2010). Software code review by team members, however, typically follows well-established practice, with careful testing of software products early in the development cycle (Thongtanunam et al. 2016). The most formal realization of this type of code review involves developers meeting and reviewing printed-out code, line by line. Such review is impractical for large open source projects.

Instead, many open source software projects follow what has been dubbed Modern Code Review (MCR), and is often called a lightweight code review. This type of code review dominates in practice (Beller et al. 2014). While identifying defects is a core goal of code review, modern code review now also seeks to transfer knowledge, increase awareness, and create of alternative solutions to problems as well (McIntosh et al. 2016). For scientific research, code review has been proposed to be become an important feature of the peer-review

process, to the extent that some authors propose that all source code for statistical analyses must be open during peer review (Shamir et al. 2013).

GitHub (see below) and other online software repositories allow collaborative code review. The examples illustrated above (`Node.js`, `vegan`, and `ActivitySim`) all feature some way of evaluating code from the community of developers. The key elements of code review are identifying the community members who will carry out the review, having clear standards for what merits useful code to accept, and having a open record of what features are currently desired or being developed.

Identifying the members to carry out the code review is crucial. For projects with clear governance, this should be straightforward, as they will be identified already as Collaborators (`Node.js`) or Committers (`ActivitySim`). For VisionEval, this may be the Contributor Review Team or another set of users they designate.

However the reviewers are designated, there should be clear standards for what code will be accepted. A bare minimum standard is that new features or modules should not break the master; this can be accomplished by automating code testing with tools such as Travis-CI. Beyond this minimum test, ideally there will be a "best practices" guide for developers. For example, the astronomy and astrophysics modeling package AstroPy provides an extensive guide for developers here. This guide covers stylistic issues of coding as well as details for how to test code and initiate pull requests on GitHub.

Finally, there will be a list of features which are desired for a project, perhaps with specific features deemed necessary for the next release. AstroPy maintains a formalized list of Proposals for Enhancements on their wiki, and smaller scale list of features to address for specific releases are maintained on the (issues page](https://github.com/astropy/astropy/milestone/49). Having these clear markers for what is desired will minimize duplicate efforts and motivate developers to contribute.

VisionEval should develop a clear work flow for code review, by designating a code review team, establishing standards for code, and publicizing desired features in order to solicit contributions.

4. Git and GitHub

Git

Git is a distributed version control system. Git is *distributed* because every user has a complete copy of the repository. Git is a *version control system* in that it tracks the changes to files, allowing useful changes to be incorporated with attribution to the authors.

At the most basic level, Git manages version control for plain-text files, by tracking changes made at the level of each character in the file. For an individual developer working independently, this feature is already useful, replacing the manual 'version control' commonly done by adding "v2" or the date to a file name. In Git, the most current version of each file is kept, and a snapshot of the differences with previous versions is also kept. Those difference files allow users to examine previous versions of files.

Other version control software, such as Apache Subversion (SVN), achieve this type of tracking changes for individual files as well. The major difference between Git and other version control systems is the distributed aspect, as opposed to a centralized system. In a centralized version control system like SVN, there is a single master copy of a project. Each developer has to connect to that master copy, make changes to files individually, and then submit it back to the master copy. While one person is working on a file, no other user can edit it. For a small group working separately on a project, SVN and similar systems serve very well.

In a decentralized system, anybody can make a copy of the working version of a project ("clone" or "fork" it), work offline or otherwise disconnected from any central server. Multiple users can simultaneously work on projects, since each copy on users' computers is a complete version of the project. If a user has made some changes which they think are of interest to others, they can request that their changes be pulled back to the original. Git suits open-source projects very well, because of this decentralized system, and promotes collaboration among users from different organizations.

Other distributed version control systems exists, such as Bazaar and Mercurial. These have fewer users, but share many of the same features as Git. A major difference between Git and other systems is the number of users and the online platform for collaborating, GitHub.

GitHub

GitHub is the Internet platform supporting the use of Git. Git itself does not require the use of GitHub, as it can be used to track file versions on an a single computer. However, in nearly all instances users of Git are collaborating with others via GitHub. GitHub promotes collaboration by facilitating cloning repositories and creating pull requests (see below) to submit project improvements. The other major collaboration tools on GitHub are the presence of a wiki and the issue-tracking tools. Since repositories exist both on the machines of users and on GitHub servers, GitHub can be considered a form of file backup, but does not replace regular backups.

Issue-tracking can be a powerful way for projects to engage users. The issue-tracking on GitHub provides a means for users to submit bugs or features easily. Users submitting issues do not necessarily need to be developers with the time and ability to propose new code which solves those issues, but can simply be users interested in improving the project. For example, an open-source data visualization project called Shiny has an active issue tracking page, with some issues attracting multiple comments from other users, and discussion with the development team. In this case, the project is centralized in governance, with a software development company curating the project with input from users worldwide.

The wiki feature on GitHub is another tool which can both engage users and promote collaboration. Wikis can serve as the complete documentation for a project, at a detailed level. For example, the data visualization project d3 has an extensive and well-curated wiki, which provides a detailed introduction to the project and links to tutorials, examples, and active user groups. Like other collaboratively modified web pages, the GitHub wiki tool allows collaborators to make page improvements. However, unlike public wikis such as Wikipedia, collaborators must be identified by the repository owner to be given permission to make edits.

Pull requests

Git development is also called pull-based development. The "pull" refers to pulling changes from individual users. On GitHub, the original creator of a repository is the 'owner', and by default the owner is the only user with the ability to merge pull requests from other contributors. This type of collaboration suits open source projects well, as occasional contributions from users can be vetted by the project team, and accepted, rejected, or have further changes proposed.

This code review process is done on GitHub. While the core team is developing a project, at any time a user can fork a repository, make edits, and then create a pull request. The core team then inspects the changes made by the pull request, and can discuss on GitHub the merits of what the changes would accomplish. In some cases, the core team (composed of the repository owner and collaborators) may ask users to make further changes to their code before accepting the pull requests. On acceptance, the code from the user is merged into project.

Only a small proportion of projects on GitHub use pull requests, with 14% of project as of 2016 found to have active pull requests (Gousios, Pinzger, and Deursen 2014). Core team members can make changes by directly committing to the project, and many projects are driven by the core team only. These projects may still be forked and used by others, but are not attracting suggestions for improvement from the user community. The majority of pull requests on GitHub projects are for small changes (10 lines of code), attract a small number of comments, and are made by developers who have made a few dozen other pull requests across all GitHub projects. Most pull requests are merged with 4 days of creation (Gousios, Pinzger, and Deursen 2014), demonstrating the rapid pace of development supported by GitHub.

Documenting changes

Documentation of open source projects occurs at both the small and large scale. At the small scale, every change made by commits to a project or via pull requests need to be documented. Every commit on Git requires a short message to document the specific change, and pull requests are documented by the Pull Requests tab on a GitHub page. At a larger scale, developers need to provide documentation about how to use the tools, which is done in both a brief README file and ideally in greater detail on the GitHub wiki. For some projects, a separate PDF file may exist as the project documentation, but such a static file may not serve the needs of an actively-developing project. For active projects, a wiki should serve as the complete documentation.

Major updates to projects may be documented by creating a release, which has a version number and brief notes to document the significant changes. Releases provide a way to mark major changes to the software tools. Since users are not downloading executable files in the traditional method of releasing software, the GitHub releases are simply milestones marking points that the core team has decided are significant changes, and can provide static snapshots of the project at that time. An example of an R package of analysis tools used by biologists here shows how the source code for specific releases is bundled into archive files, with extensive documentation.

Communicating with the user community

Successful projects have a community of users who employ and improve the tools of the project. Users can become engaged in the project by contributing suggestions via the issue-tracking tool on GitHub, or directly contributing project improvements with pull requests. However, one common method of engaging users not supported by GitHub is a forum for users to ask general questions and share knowledge. Such forums include distribution platforms such as the Open Source Application Development Portal of the Federal Highway Administration, or individual project web pages, such as the UrbanSim project for planning and analysis of urban development. User forums of a sort also exist outside of official project web pages, for example on the highly active Stack Overflow site. This site which allows developers to ask and answer questions about specific problems with a tool, or about issues more general than one tool. A forum can be developed for a GitHub Pages site, which is a way to host a front-end website for a GitHub project.

For projects such as VisionEval which will likely have a small number of specialized users, a discussion forum may not be the most beneficial tool. Instead, using Issues on GitHub likely will present the best way for users to discuss specific technical topics about VisionEval tools. As users of VisionEval tools will be technically sophisticated enough to know about and use GitHub, keeping the online discussion on the GitHub pages (namely, the Issue tracker) will be the most convenient way to host discussions.

In addition to online discussions, periodic announcements by email can be useful. Such announcements may be on at regular periods, or following releases of major updates or new modules. Maintaining a list of interested users would require some investment of effort for the centralized governing organization. Large open source projects with many active users do not typically have email announcements, but for VisionEval such announcements could be useful for engaging the community of users who are not developers.

5. Recommendations

As an ongoing open source development effort, VisionEval has already implemented key best practices, namely developing the software in an open, public fashion with documentation of the contributions being made. Making VisionEval a vibrant community of users and developers will require a few additional steps. These include:

- Agree on a license which supports the goals of the project. The VisionEval team should consider as permissive a license as possible, so that the public may benefit from these strategic planning tools. Patent protection and the need to have releases from individual developers (CLAs) should be considered.

- Develop and document governance procedures to define the roles and responsibilities of users, developers, and collaborators/committers, as well as an oversight body composed of funders. Following the ActivitySim structure would appear to be a useful way forward.
- Develop a code review process which is clear and fair, to maximize contributions from the community. This will be likely done on GitHub, using pull requests to merge new features into the "review" or testing branch, with periodic merges into the master branch when warranted to release a new version.
- Clearly document the software tools, to grow the community of users and developers. Guidance for developers should be a priority, including conventions about how to comment code and how new functions will interact with existing functions. A project wiki on GitHub (or one for each of the VisionEval models) would be recommended. Mailing lists for users and developers may not be necessary at first, as the issues page on GitHub can serve most of the needs for communicating. A project webpage (https://pages.github.com/) may be useful, to provide a simple interface for non-developer users.
- Release stable versions periodically. Releases can be announced by email if an email list is developed, or simply announced on the GitHub page.

Taking these steps will help grow the community of users and developers to collaboratively develop the VisionEval strategic planning tools.

References

Aksulu, Altay, and Michael Wade. 2010. "A Comprehensive Review and Synthesis of Open Source Research." *Journal of the Association for Information Systems* 11 (11). Association for Information Systems: 576.

Beller, Moritz, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. "Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix?" In *Proceedings of the 11th Working Conference on Mining Software Repositories*, 202–11. ACM.

Bettenburg, Nicolas, Ahmed E Hassan, Bram Adams, and Daniel M German. 2015. "Management of Community Contributions." *Empirical Software Engineering* 20 (1). Springer: 252–89.

Chen, Daoyi, Shahriar Shams, César Carmona-Moreno, and Andrea Leone. 2010. "Assessment of Open Source Gis Software for Water Resources Management in Developing Countries." *Journal of Hydro-Environment Research* 4 (3). Elsevier: 253–64.

Gousios, Georgios, Martin Pinzger, and Arie van Deursen. 2014. "An Exploratory Study of the Pull-Based Software Development Model." In *Proceedings of the 36th International Conference on Software Engineering*, 345–55. ACM.

Heistermann, Maik, S Collis, MJ Dixon, S Giangrande, JJ Helmus, B Kelley, J Koistinen, et al. 2015. "The Emergence of Open-Source Software for the Weather Radar Community." *Bulletin of the American Meteorological Society* 96 (1): 117–28.

McIntosh, Shane, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. "An Empirical Study of the Impact of Modern Code Review Practices on Software Quality." *Empirical Software Engineering* 21 (5). Springer: 2146–89.

Raymond, Eric. 1999. "The Cathedral and the Bazaar." *Philosophy & Technology* 12 (3). Springer: 23.

Shamir, Lior, John F Wallin, Alice Allen, Bruce Berriman, Peter Teuben, Robert J Nemiroff, Jessica Mink, Robert J Hanisch, and Kimberly DuPrie. 2013. "Practices in Source Code Sharing in Astrophysics." *Astronomy and Computing* 1. Elsevier: 54–58.

Thongtanunam, Patanamon, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2016. "Review Participation in Modern Code Review." *Empirical Software Engineering*. Springer, 1–50.

Von Krogh, Georg. 2003. "Open-Source Software Development." *MIT Sloan Management Review* 44 (3).

Massachusetts Institute of Technology, Cambridge, MA: 14.

Von Krogh, Georg, and Sebastian Spaeth. 2007. "The Open Source Software Phenomenon: Characteristics That Promote Research." *The Journal of Strategic Information Systems* 16 (3). Elsevier: 236–53.

Wilson, Jennifer AND Cranston, Greg AND Bryan. 2017. "Good Enough Practices in Scientific Computing." *PLOS Computational Biology* 13 (6). Public Library of Science: 1–20. doi:10.1371/journal.pcbi.1005510.

Zhu, Kevin Xiaoguo, and Zach Zhizhong Zhou. 2012. "Research Note—Lock-in Strategy in Software Competition: Open-Source Software Vs. Proprietary Software." *Information Systems Research* 23 (2). INFORMS: 536–45.