The sustainability of scientific software: ecosystem context and science policy

James Howison
University of Texas at Austin
UTA 5.404
1616 Guadalupe Street
Austin, TX 78701
+1 315 395 4056
jhowison@ischool.utexas.edu
Corresponding Author

James D Herbsleb
Carnegie Mellon University
Pittsburgh, PA
jdh@cs.cmu.edu

Abstract

The sustainability of scientific software is a key challenge for science policy. We approach this question by drawing on empirical studies of scientists using software and describe how components are arranged with complements and dependencies into value-providing assemblies, periodically revisited by their scientist users. Over time, software declines in scientific usefulness, driven by four factors: a moving scientific frontier and technological change, production friction, use friction and the software ecosystem context. In particular we highlight the impact of the complexity of ecosystem context, in terms of the diversity of use-contexts in which a component is used. We identify three broad strategies to address the need for work to sustain the usefulness of scientific software: suppress the drivers, reduce the amount of work needed, or attract sufficient resources able to undertake the work needed to sustain scientific usefulness. We examine three attraction systems: commercial markets, community-based peer-production and grant-making. We describe how these systems bring resources to projects, and particularly highlight how both commercial markets and peer production address the challenges of ecosystem complexity while scientific grant-making does not. We conclude by making science policy recommendations to address the challenges of sustainability, by enhancing the grant-making system and by facilitating transitions to other resource attraction systems.

Keywords: science policy, software development, innovation ecosystem, peer production

1 Introduction

Science depends on software. From configuration and control of instruments, to statistical analysis, simulation and visualization, virtually every workflow that generates scientific results involves software. Recent research suggests that scientists may be spending up to 30% of their time developing software and 40% of their time using software (Hannay et al., 2009). Indeed, in many fields there is no scientific data without simulation models realized in software (Edwards, 2010).

Visions of the future of science, such as the Atkin's Report and the NSFs CIF21 vision (Atkins, 2003; NSF Cyberinfrastructure Council, 2007), frame software as much more than a supporting service: it can be a source of innovation and can enhance science by increasing its transparency, reproducibility, correctness, transferability and scale (Ince et al., 2012; NSF, 2012; Stodden et al., 2010). In particular the vision holds that the properties of software as an information artifact, its low marginal cost of reproduction and high potential for re-use and recombination, offers the potential for relatively small initial investments that can lead to increasing re-use and coalescence into widely used software platforms, resulting in widespread, long-lived, impact in the form of better science (NSF, 2012).

Yet software can also become a problem, consuming time and resources from science, with duplicated work, poor quality results and weak reproducibility (Atkins, 2003; Carver et al., 2007; Dubois, 2005; Gambardella and Hall, 2006; McCullough et al., 2006, p. 11; Segal and Morris, 2008). Indeed questions about the quality of software and software work in science were at the heart of recent debates about the reliability of scientific results for public policy precipitated by the so-called "Climategate" incident (Reay, 2010; Ryghaug and Skjølsvold, 2010). The potential contribution of software in science is thus undermined, resulting in practices that obscure rather than reveal the underlying science (Ince et al., 2012; Stodden et al., 2010) and expensive, frustrating, churn as packages are written and discarded.

A pressing question of science policy, therefore, is how to overcome these challenges and work towards the positive vision of software in science. At a very general level these are questions about how the production and use of software in science comes

together to produce its impact as a socio-technical system (Trist and Bamforth, 1951). More specifically, this includes questions of appropriate technologies and development practices, but also questions of how software and its production and use intersects with existing institutions. While concerns about the effectiveness of IT investments are far from unique to science (Brynjolfsson, 1993; David, 1990), the specifics of science as a domain of human activity bring forward new and interesting questions, including how the different incentive systems involved in science structure work and innovation (e.g., David, 2002; Howison and Herbsleb, 2011; Huang and Murray, 2010; Riggs and von Hippel, 1994). In particular there is a growing realization that software is different from other scientific results because software, unlike publications, has substantial ongoing costs if it is to remain scientifically useful, a pre-condition to achieving the hoped for gains to science, such as improved transparency, correctness and innovativeness. Indeed one of the very few empirical studies of ongoing software work in science concludes that the ongoing work looks very similar to initial development work (Bietz et al., 2012), a finding that echoes the growing emphasis on post-development costs across the software industry (e.g., Boehm et al., 1995).

Further, software is almost always composed from multiple components. This raises questions about how components and their different production systems interact, approaching scientific software in a manner analogous to an innovation ecosystem (Adner and Kapoor, 2010; Jansen et al., 2013; Messerschmitt and Szyperski, 2005) and questions about how technological structures interact with policy and strategy (Baldwin and Clark, 2006, 2000; MacCormack et al., 2006). Finally, and practically, the challenge of scientific software raises questions of science policy, from what outcomes to prefer and what policy levers are appropriate for guiding activity towards those outcomes.

Scientific policy makers are aware of these opportunities and issues and contemplating policy responses. The NSF has organized two workshops in recent years, "Software Sustainability through Investment" (Alexander, 2009), "Cyberinfrastructure Software Sustainability" (Stewart et al., 2010) and a 2013 workshop at the Supercomputing conference (SC13) "Working towards Sustainable Software for Science: Practice and Experiences," to consider challenges and solutions for sustainability in scientific software.

In this paper we draw on existing empirical research to characterize the problem of the sustainability of scientific software, and provide a framework for analyzing the issues in a way that provides science policy recommendations.

1.1 Sustainability of what? Revisiting the goal.

From an innovation and science policy perspective, scientific software is valuable to the extent that it ultimately advances the practice of science, contributing to the stock of knowledge that underlies modern societies and economies. For this reason, we argue that it is most useful to begin by understanding how it is that software is used by scientists and how time affects the scientific value that software provides.

We draw on results from a qualitative study of software in science, reported in detail in Howison and Herbsleb (2011). **Figure 1**, shows the results of reconstructing scientific software use. The reconstruction began with three high-quality papers in different fields (high-energy physics, micro-biology and structural biology) and, drawing on semi-structured interviews, built a narrative of how the science was undertaken, focusing on the role of software. The empirical work drew on the published articles, methods and materials sections and interviews with the authors and research staff involved in the science, identifying internal documents and source code produced in the course of the work.

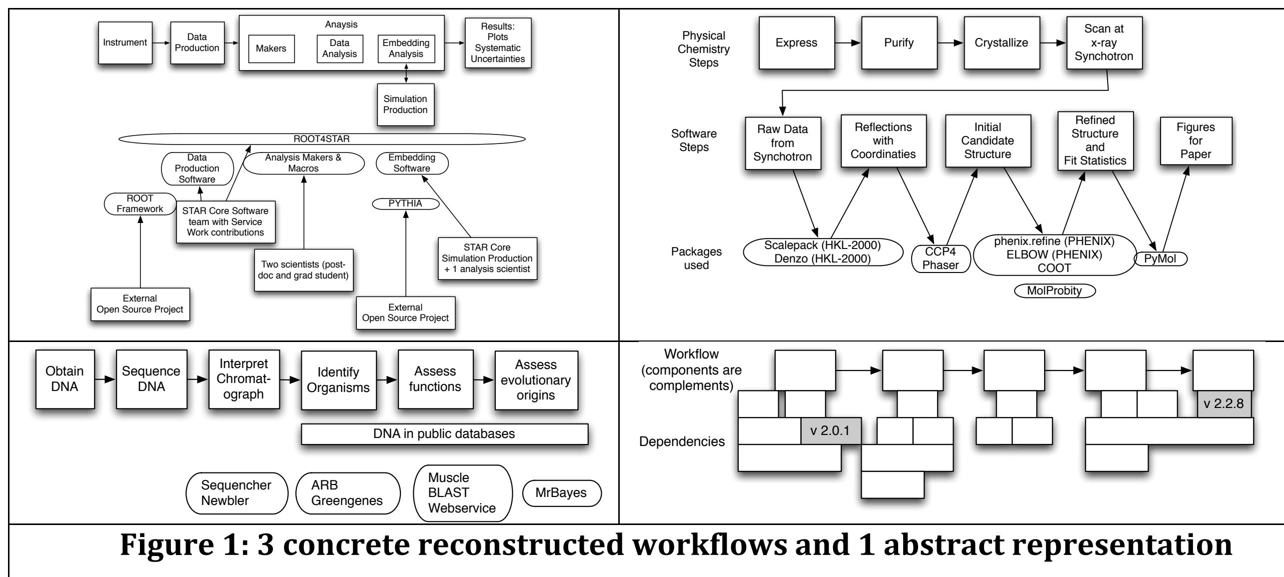

Figure 1: 3 concrete reconstructed workflows and 1 abstract representation

In this way the research identified all the software used in the production of each paper, arranging the software according to the workflow of the science undertaken. This workflow grounded a narrative describing the software work underlying the science. Inspired by the literature on innovation and software ecosystems (Adner and Kapoor, 2010; Iansiti and Levien, 2004; Messerschmitt and Szyperski, 2005), Howison and Herbsleb described how and why each piece of software was produced and what role it played in the science. They began with software written by the authors of the focal papers and extended outwards to the production of software used by the authors then further outwards the production of the software on which each piece of software depends. They grew this production web until encountering general purpose software whose production was not driven by its use in science, such as word processors and operating systems.

One takeaway from these reconstructions was the location of any particular component of scientific software within a matrix of both complements (horizontal in **Figure 1**) and dependencies (vertical in **Figure 1**). Complements are those components which undertake other aspects of that workflow, such that the high-level components together provide the collection of information processing needed to generate the scientific result. Dependencies provide the services which components higher up the stack utilize, either to provide still more services to yet higher level components or, ultimately, to carry out steps of a scientific workflow. While complements are often quite visible to the scientist who has likely handled that software themselves, dependencies are often incorporated indirectly, brought into place by one of the complements.

In the parlance of software practice, the top-level horizontal view is typically called a workflow (e.g., Gil et al., 2007), while the vertical view is popularly called a “software stack."[1] The arrangement of multiple elements into software has been referred to in the academic literature as the "software architecture," (Garlan and Perry, 1995; Garlan and Shaw, 1993), sparking a fruitful field of inquiry into the characteristics of software composition. Yet the work our informants described did not much resemble architecture, with the implication that the architect is aware of all the software implicated, envisions the components as a system and engages in the joint optimization of that system.

[1] http://en.wikipedia.org/wiki/Solution_stack

Software as it was used by the scientists we spoke with was not just functionality that results from a composition of functional elements, but also includes preprocessing, post-processing and presentation elements that are brought loosely together and used to accomplish a scientific purpose (Anderson et al., 2007). Accordingly we call the work that the domain scientists in our study engaged in "assembly," emphasizing the manner in which they arranged existing components, sometimes enhancing them with glue code or new, customized, components specific to the scientific problem they are addressing. Accordingly, we call the result of the scientists' work a "software assembly," made up of many existing components and perhaps one or two custom components brought together through mixed scientific and software work.

1.2 Scientific assemblies over time

In one sense the scientist's software assembly exists to execute and support the specific piece of science embodied in a specific paper. Yet, over time, scientists revisit their assemblies, seeking to re-use them as they push forward the frontier of their science. While our informants often used the word "replication," in practice they revisited their assemblies to apply them to new data and to enhance them with changes, perhaps the addition of a new processing step or using a new, better, algorithm in place of one used before. The scientific frontier moves forward and the software assembly must move with it.

Revisiting software assemblies raises the question of what happens to these assemblies over time. The clearest, but trivial, answer is nothing. At their core software assemblies are compilations of 1s and 0s and given the trivial resources of electricity and disk storage they can exist indefinitely as they were when first assembled.

Yet, in practice, the assembly's relationship to scientific work changes and its scientific usefulness declines. The scientist is not just running the code, but seeking to *work* with the code. The desire to work with the code drives the scientist towards current, updated, versions of complements, if not to take advantage of new knowledge reflecting recent scientific advances embodied in new versions of software, at least to take advantage of new features, better performance, new hardware support, the possibility of relevant documentation and the availability of support, either from the developers or the user community.

This dynamic is quite apart from the difficulties of even running old code in the first place. In practice, it appears that scientists are rarely aware of the software assembly as a whole, due to the indirect incorporation of dependencies. Rather they focus on the portions considered most primary, essential or novel. Thus storage may take the form of storing the full horizontal workflow or, more frequently, it may take the form of storing only bespoke components and plumbing work and listing complements. In many cases, storage is quite possibly more the result of inaction rather than action, with the scientist having simply left the files where they were when the science was conducted. After all, the paper is published and the scientist's attention is elsewhere.

Thus, for a software assembly to be re-visited and worked on, it must be laid out with all complements and "on top of" its dependencies. The work of re-animating the assembly, even those parts the scientist does not want to change, requires a sort of "software archeology"[2] whereby the appropriate dependencies are identified, located and placed into service. Since dependencies were often implicit and invisible at the workflow level even identifying dependencies can involve a recursive and frustrating process of reading the "build files" of each component, translating from various barely human-readable formats and puzzling out their implications. The scientist, or more likely, the graduate student, undertaking this work will typically find that the components they are seeking have themselves changed over time, as discussed below, requiring them to trade off the work of obtaining historical versions and getting them to work, against the work of adapting surrounding components to work with newer versions. It is far from uncommon to discover what appear to be cyclic dependencies, to require missing historical versions, or to require multiple, incompatible versions, requiring some level of jerry-rigging at points in the stack. This experience is common to those working with software, even outside science, and is described as a descent into "dependency hell."[3]

Even in the best of circumstances, then, the work of extracting ongoing scientific value from software requires considerable work for the scientists. Even if the rest of the software ecosystem had stood still, the moving frontier of science and the opportunities afforded by new hardware build in dynamism at the edge.

---

[2] http://en.wikipedia.org/wiki/Software_archaeology
[3] http://en.wikipedia.org/wiki/Dependency_hell

But of course, the software ecosystem does not stand still; scientists revisiting assemblies find that the components themselves have changed, often rapidly and in ways that require extra work from end-user scientists (see also Bietz et al., 2012, 2010). To understand why, we must move our focus from the scientists preparing a particular paper to the projects producing the components the scientist is assembling.

From the perspective of component producing projects, matters are both similar and different. In one important sense any component is likely to itself be a software assembly: the component has both dependencies and internal complements (external components that participate in the flow of computing that the component as a whole produces). Further, like end-users, component producers are themselves subject to the same pressures that drive forward work at the edge. They must manage changing opportunities offered by new hardware or execution architectures. They too are motivated to improve their software, extracting greater or more reliable performance. Moreover, many component producers themselves participate in the scientific reputation economy, seeking to publish papers describing the advance of their tools, or obtain new grants based on feature extension (Howison and Herbsleb, 2011). In short, the component producer's own scientific frontier moves forward, driving a need for novelty and progress.

On the other hand, matters at a producing project are different. A project producing code that others use, unlike an end-user scientist, has its artifacts passing into the scientific practice of others. A component producing project has to not only produce potentially useful software, but help its users realize that usefulness, supporting their use by documenting code, providing examples and tutorials and, inevitably, answering questions.

And there may, of course, be many users. Thus the component might play a role in a many different assemblies, interacting with many sets of data, complements and dependencies. Some of these assemblies might be relatively similar, while some might be quite different, such that the component can be arranged with different complements and dependencies, or perhaps even more complexly, occupy broadly similar but subtly different positions. Moreover each of these assemblies is being constructed and reconstructed at different points in time and changing at different paces, driven by the changing scientific frontier and work rhythms of its scientist users.

The image that presents itself (if we might be allowed considerable poetic license), pulling back to consider a wide-lens view of all the software assemblies at once, is one of components brimming with dynamic potential, vibrating in place, moving sideways to make room for new complements, shifting downwards as new components build on their capabilities, sometimes jumping to other kinds of assemblies entirely. Such a complex system is never in stasis, nor does it change in regular or predictable patterns. To the extent that it supports scientific work at all, it does so because of people's work on, with, and around components, continually re-shaping them so they are scientifically useful in a particular position in a particular arrangement at a particular time.

If the work is not done, things break down. Components cease to merely vibrate in their many assemblies but begin (if we might be allowed a continuation of our poetic license), to rattle, to shake, to expel themselves from their place, like cogs flying free of a machine.

Yet scientists are still driven to do their science and their work does not stop. Rather, a tension builds up around an assembly, frustrating its users and generating motivation to find a different component that fits this changed niche. Either the scientist themselves, or their grad student or post-doc, writes a new component more or less ideally fitted to this problematic hole in the assembly, or the need is so widely felt that a new project emerges to fill the hole with a new component. The component spreads out into other assemblies, both end-user and component-producing, sparking new rounds of adaptation and adjustment as end-users revisit their assemblies. As time moves forward, assemblies continue to change shape and this new component itself, once ideally-fitting, begins to vibrate, perhaps even to rattle a little, requiring its own work to sustain its usefulness.

Sustaining the scientific usefulness of software in our illustration above, is above all a matter of work. Accordingly, we argue for a definition of sustainability as the condition that results when the work needed to keep software scientifically useful is undertaken. The work takes many forms, from assessing and meeting new scientific or hardware opportunities, adjusting and adapting components, and supporting users. Of course, while we have not emphasized it above, producing software itself involves significant work, from understanding what to build, gathering the resources to attract team members,

coordinating development in sometimes far-flung teams and integrating contributions; even distributing new versions of software to others is significant work.

To realize the potential of sustained innovation envisioned in the cyberinfrastructure vision we argue that the need for work of all these kinds must be addressed. These needs can be addressed in three main ways: 1) suppressing the causes that drive the need for work, 2) reducing the effort needed to do the work, or 3) attracting and retaining the resources needed to do the work.

We now turn to lay out more formally the factors that drive the need for different kinds of work, focusing in particular on the impact of ecosystem context. We then characterize the broad strategies for addressing these needs, focusing in particular on the capabilities of different resource attraction systems to face the challenges of particular ecosystem contexts. We conclude by recommending appropriate science policy responses aimed at improving the effectiveness and efficiency of the scientific software ecosystem to in supporting science.

2 What drives the need for different kinds of work?

If the scientific usefulness of software is to be sustained, four drivers of needed work must be addressed: 1) exogenous drivers, 2) production friction, 3) use friction, and 4) ecosystem context.

2.1 Exogenous drivers

Two key drivers that require work are 1) progress in science and 2) changes in underlying technologies. The progress of the scientific frontier throws up new questions, data and approaches, both within fields already heavy in computation and as fields develop computational methods. The progress of science is uneven and extremely difficult to predict. Scientific opportunities are also urgent, linked as they are to the opportunity for scientific priority and the reputational rewards that come with it.

The invention of new computational technology also plays a role in creating a need for work in scientific software. This is clearest when a new generation of hardware technology arrives, such as the development and spread of parallel computational

architectures, the widespread availability of specialized GPUs, or the ubiquity of mobile computing. These underlying changes create opportunities to exploit new performance capabilities. In some cases they reduce the cost or time of computation in a manner which passes a threshold and brings techniques previously too expensive within the bounds of possibility.

2.2 Production Friction

A clear set of factors driving work in scientific software are linked to the production of software. In short the production of code is not a simple, smooth process, but requires significant knowledge and effort to execute successfully. This is true whether we consider the initial production of novel software components, or on-going improvement or adjustments. Even if the rest of the drivers of work are held constant, it is difficult to know what to build, to create a design that meets requirements, to realize it in code and to test its performance. Since development projects are often large enough to be beyond the capabilities of individuals, additional friction derives from working in teams (even if all resources are committed): designing appropriate task breakdowns, managing interdependencies, integrating contributions as well as managing conflict and providing leaderships.

2.3 Use Friction

If code is to become widely useful in science, the code must find its way from its production environment to scientist users, be assembled with dependencies and complements, and be configured appropriately. As with production work, this is far from a smooth path. Software must be released: it must be packaged and made available to users for download, both initially and for new versions. If users are to use the code effectively they must come to understand its potential, its operation and its limitations, both initially and with on-going releases. As described above any component must be arranged with complements and dependencies in order that it do scientifically useful work. This means understanding interactions between components, often in situations unanticipated by the component's producers.

2.4 Ecosystem complexity

A fourth driver of work derives from the complexity with which components are arrayed by scientific end-users and the frequency and rhythm of change of those components. To understand this driver, we consider an aggregated view of science end-user assemblies and to consider the position of individual components within that aggregation. The relationships of use, complementarity and dependency form a complex web which has been referred to as a software ecosystem (Adner and Kapoor, 2010; Jansen et al., 2013; Messerschmitt and Szyperski, 2005).

We argue that the manner in which a scientific software system drives the need for work can be understood by drawing on two dimensions: 1) users, a simple count of the number of assemblies that a component appears in, and 2) use-contexts, the number of different positions that a component appears in across assemblies. Use-contexts are loosely related to scientific fields, but since different fields can use components with similar complements and dependencies, and different scientists within a field can array the same component differently, the relevant context is not the discipline of the scientists, but the "neighborhood" of components with which a component is arranged.

We refer to the combination of these dimensions as ecosystem context. Figure 2 illustrates this, with number of users on the vertical and number of different use-contexts on the horizontal. We show the top left as generally unreachable, because although some use-contexts might have a relatively large pool of potential users, others will only have a small pool, thus in general the highest potential number of users can only come by moving rightward, implying a larger number of use-contexts. Similarly the bottom right, reflecting a high number of use-contexts but a low number of users, is practically unlikely because each different use-context implies use in a different software assembly, implying at least one user per assembly. Nonetheless, there is a wide variety of ecosystem contexts available: from the bottom left of a component with only a single user (and thus a single use-context), to the middle-left of a component with only a few use-contexts, but each with multiple users, to the upper-right, reflecting a component arrayed in many different use-contexts and having many users. Each of these dimensions is associated with a different balance of production and use friction and thus needed types of work.

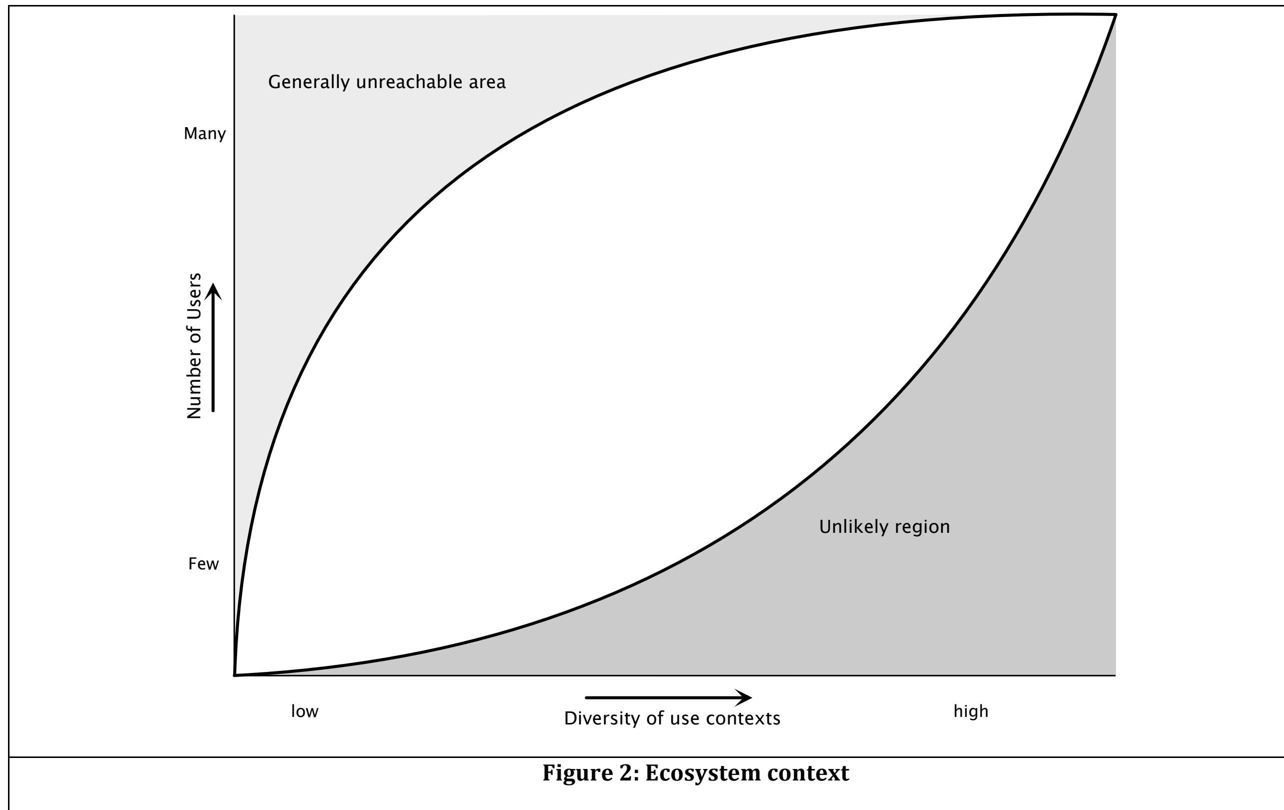

Figure 2: Ecosystem context

2.4.1 Greater user numbers

As one moves vertically and considers components with higher numbers of users—but each with similar use-contexts—production friction stays relatively constant while use friction rises. Production friction stays close to constant because the solutions and artifacts, once found, are available for distribution to all users and are useful for all users. Requirements work can be done with any single user and the project must only monitor and learn of changes to a single set of complements and dependencies.

Use friction, on the other hand, rises with the number of users, each of whom must come to know how to make use of the component, obtain, and array complements and dependencies. More users, even with identical use-contexts, brings with it more questions that must be answered; this is especially true because users, even with identical use-contexts, are likely to be at different places on their learning curves. Yet because use-contexts are similar, as with solutions to production problems, solutions to use problems are more likely to be re-usable. In this way documentation useful to one user is likely to be

useful to others, and answers of questions for an early stage user are likely to remain useful as other users, new to the component, begin their use.

2.4.2 Greater numbers of use-contexts

The dimension of use-context is associated with a different balance of production and use friction. As one moves horizontally and considers components with a greater diversity of use-contexts, production friction rises, while use friction remains relatively constant. Production friction rises because each use context implies a different source of change through the complements and dependencies with which a component is arranged. When a neighboring component changes, there is a need to understand those changes, assess whether a response is needed and to produce the relevant changes to retain the scientific usefulness of an assembly. As we will discuss below, this work can be (and often is) done at different places in the ecosystem, including by the end-users or the producing project. If it is done by other than the end-users then each adjustment also implies more production friction, in the form of packaging, releasing, and distributing the relevant changes.

If a change in a single surrounding component drives a need for production work, change occurring across the variety of use-contexts implies a rapid increase in production work; indeed because this work is driven by the combination of components and the solutions produced are not necessarily re-usable, or even compatible, the increase in the need for production work is super-linear as the diversity of use-contexts rises.

The frequency and rhythm of change in surrounding components can also drive a need for work. Frequency matters because each individual change in a neighboring component implies a round of assessing, adjusting and, perhaps, distributing changes. Therefore frequency of change acts similarly to changes in the scientific frontier and underlying technologies, injecting new needs for adjustment work. The more frequent the changes, the more work that needs to be done to keep a component scientifically useful.

The rhythm of change can also be important. This is because adjustments take time to spread through the network of dependencies and out to end-users. At some point new changes could be occurring before the adjustments to the last changes have spread throughout the network, especially if changes occur close together in time. This leaves

some users working with older versions, complicating user support and adjustment to changes in surrounding components. If the exogenous needs for change are pebbles dropped in a pond, the impact of rhythm and pace can be thought of as ripples catching up with and over-taking each other; adjustments originating in the same place, but at different times and traveling through scientist's software assemblies at different speeds.

The impact of the frequency and rhythm of change depends on the ecosystem context, in terms of number of users and diversity of use-contexts. This is best understood considering a project monitoring how its component is being used and undertaking production work to adjust for changes in its use-context. If that component has many users, but they are all using the software identically, then a change in the use-context is relevant to all the users at the same time. Provided the component producing project hears of and understands the change, a solution can be produced and distributed to all users. The frequency and rhythm of changes may vary, but the requirements occur at the same time, and the adjustments are relevant to all users at the same time.

However, when use-contexts vary, adjustments may be called for by many different components at the same time. Moreover each adjustment may be relevant to only some of the use-contexts, and not to others (or worse, an obvious solution for one use-context might be incompatible with others). This can be illustrated as dropping multiple rocks into a pond in different areas at different intervals; as the ripples move outward they begin to overlap and interact, crisscrossing or perhaps doubling-up.

Finally the already complex situation can be further exacerbated because adjustment work does not only occur at a component producing project, but can also occur at scientific end-use points. This might occur because the end-users are under deadlines, do not understand the origins of different components, or do not relish interacting with (and thus being dependent on) other groups to manage the changes they perceive in surrounding components (e.g., Howison and Herbsleb, 2011). Such changes can quickly lead to bifurcations in use-contexts between user assemblies (changes in a component's neighborhood) and a component producing project can, without their knowledge, move from having few use-contexts to the much more complex situation of having many use-contexts. Worse, as use-contexts multiply the demands on a component producing project

to collect and respond to changes rise, potentially creating additional delays that lead to further end-user adjustments and exponentially exacerbate the problem.

Table 1 shows the kinds of work called for by these different drivers.

Production Work	
Design	Deciding what to build
Development	Building the design
Integration	Adding new contributions to the existing codebase
Release work	Making code available
Management work	Coordinating contributors
User Support	
Documentation	Providing non-code resources to explain software use
Answering questions	Helping users by answering specific, contextual, questions
Ecosystem Work	
Sensing	Observing use-contexts to see changes in surrounding components
Adaptation	Adjusting components to continue to work
Synchronization	Collecting adjustments for release to avoid cascading re-adjustment

Table 1: Different kinds of work required to sustain the scientific usefulness of software

3 What can be done about the needed work?

Sustainability in scientific software is a problem because the four factors outlined above require ongoing work to ensure the ongoing scientific usefulness of software. In this section we outline three broad strategies to addressing these needs. The three broad strategies are 1) to suppress the factors driving the need for work, 2) to reduce the amount of work needed, and 3) to attract resources willing and able to do the work needed. In the following sections we consider the feasibility and realization of each strategy in science.

3.1 Suppressing the factors driving the need for work

The simplest way to address a need for work is to suppress the factor driving that need.

3.1.1 Suppression of exogenous drivers

An obvious initial strategy is to avoid the issue entirely by not using software at all, thus obviating the impact of all other factors. A second, less severe, strategy would be to

reduce or eliminate change resulting from the progress of science or the production and introduction of new technologies.

Indeed, analogies to these approaches are indeed used outside science. For example the music business is notorious for seeking to resist the introduction of disruptive new technologies or approaches, or at least to slow their introduction while they adjust other aspects of the industry (such as streaming licensing agreements) to preserve the profit-making potential of the industry (e.g., Leyshon, 2001). This includes efforts to synchronize and pace the introduction of new technologies, facilitating their saturation of the market prior to introducing the next technology.

On a less grand scale, however, suppressing the introduction of new technologies is a common technique in large organizations. By standardizing a technology across an organization and resisting the introduction of others, the organization trades some innovative potential for reduced complexity (Alt, 1964; West, 2006). A classic example is standardizing on the use of a particular language, say Java, and restricting the use of newer languages, such as Ruby. Other examples include suppressing the introduction of a new generation of a technology into an organizational ecosystem, especially until the organization as a whole has transitioned, and perhaps choosing to skip a generation (or many generations). An example would be choosing not to upgrade from Windows 95 to 98, but moving directly to Windows XP. Those responsible for strategy in the organization perceive the complexity of cascading adjustments needed, choose to suppress a cause, and shoulder the frustration at the edges that commonly results from such policies.

Within science, of course, this strategy is problematic not only because of the high valuation of innovation, but also impractical because of the lack of centralized decision-makers with appropriate insight and legitimacy to command and enforce any suppression oriented strategies. Further the periodic, rather than continuous, nature of scientific end-use reduces the value to be derived from exploitation of existing technologies (which have aged and possibly de-synchronized in the time elapsed between revisiting software assembles) and prompts efforts to be “up to date.” Nonetheless, suppression strategies can be and are enforced more locally, such as within individual labs, centers or scientific collaborations.

3.1.2 Suppress the impact of complexity of the ecosystem context

A second set of strategies focuses on the work requirements driven by ecosystem complexity. As argued above, the need for sensing, adjustment and synchronization work is primarily driven by the diversity of use-contexts with which a component is arrayed. Yet the impact of changes in these diverse use-contexts and the route that impact passes through the ecosystem can be affected by the overall ecosystem structure, particularly the creation of layered and platform architectures (Baldwin and Woodard, 2009; Baldwin and Clark, 2000; Boudreau, 2010; Garlan and Shaw, 1993; Gawer and Cusumano, 2002; Iansiti and Levien, 2004).

Figure **3** depicts three idealized ecosystem structures. In this illustration two components are connected if they are used together in at least one software assembly (whether that be as a dependency or a nearby complement), thus these diagrams are different from the software assemblies of **Figure 1**. The lines represent potential paths of change impact, transferring through the software ecosystem and generating a need for adjustment at the component. One can think of these diagrams as transmission paths, such that a change at a particular component (perhaps resulting from a change in the scientific frontier) acts like a "pulse" and can be transmitted along these lines. When that pulse reaches a connected component, adjustment work there might cause a new pulse, such that components connected to the newly adjusted component now may need to undertake adjustment work.

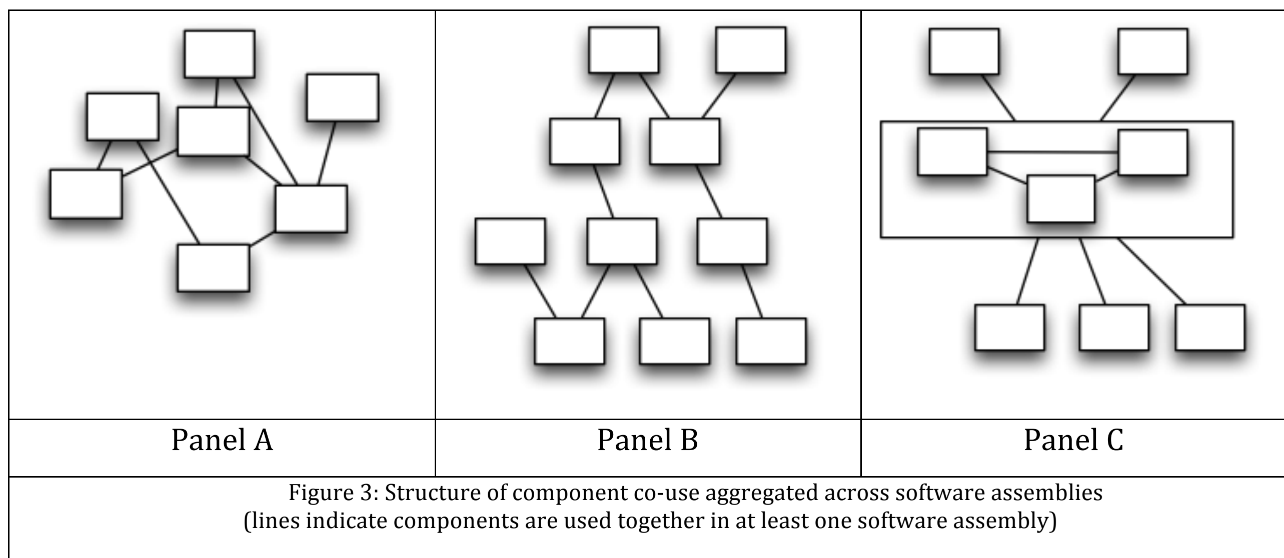

Figure 3: Structure of component co-use aggregated across software assemblies
(lines indicate components are used together in at least one software assembly)

Panel A of Figure **3** illustrates a randomly connected component graph, including relatively dense interconnection, long paths and circular connections. Changes initiated anywhere in this network can have impact across the graph, sparking resulting adjustments that cascade chaotically throughout the graph. Some components are heavily connected (reflecting use in a high diversity of use-contexts), concentrating work requirements and creating bottle-necks for adjustments. The circularity of the interconnection even raises the possibility of self-sustaining loops, where the ecosystem never completes adjusting to a cascade of changes.

Panel B depicts a hierarchical arrangement of interconnections, often known as a layered architecture. Connections follow single, hierarchical paths, excluding the possibility of circularity. A change in component requirements in such an ecosystem radiates "up-stream" until finding a position of maximal generality, before finding a route back "down-stream" to minimally connected components which are end-points for waves of adjustment. Fewer and less complex paths of interconnection restrict the impact of ecosystem complexity and minimize the needed work.

Panel C depicts a further refinement of the hierarchical structure, separating components into different types and collecting those that are densely or circuitously interconnected into a platform. The platform acts as a single large component, hiding complexity from the ecosystem. Changes from the edges collect in the platform, general solutions can be found and released in a synchronized manner, reducing cascades of adjustment. In the particularly idealized arrangement shown in Panel C, components above the platform have no connection other than with the platform. Not all platforms realize this additional constraint, for example Apple's iOS comes close (applications on a phone do not rely on services from each other) while components using the Eclipse or R platforms often draw on services provided by other, non-platform, components. Fewer interconnections outside the platform implies trading off potentially innovative recombination at the edges in order to suppress ecosystem-wide requirements for on-going sensing, adjustment and synchronization work.

Ecosystem-wide architectural patterns can be powerful in suppressing the need for work to maintain the usefulness of components. Indeed, science policy-makers are well aware of the usefulness of platform architectures, as indicated in the NSF's CIF21 software

agenda (NSF, 2012) and the popularity of the "middleware" architectural design pattern. Yet it is rarely noted that achieving rationalized architectures involves influencing the behavior of end-users, not only component producers. This is because, as we have argued above, ecosystem context derives not from design intentions at component producers but from the manner in which end-users put together components. In hierarchical organizations behavior of end-users can be enforced through top-down, directive, policies, such as that employed at Amazon by "Dread Pirate Bezos" (Yegge, 2011), requiring all components to implement a web-services interface and to use Amazon's infrastructural services. Firms selling components can enforce particular conditions on their use (as Apple does with its iOS platform, particularly through controlling access to its App Store).

Yet science policy-makers do not have directive power over scientist end-users; in fact directive control would be seen as illegitimate since that would undermine the freedom of scientists and the wellspring of innovation seen to underlie scientific progress. For example policy-makers cannot require the use of particular software components, choose not to place requirements on the use of components they have funded with other components, nor can policy-makers prevent end-users from creating (and then releasing) custom components. As a consequence, creating and maintaining rational architectures is a particular challenge; we consider options to use this strategy that are available to science policy makers in our conclusion.

3.2 Reduce the effort required to do the needed work

Efforts to suppress the drivers of work requirements can be powerful, but short of abandoning software, its re-use, or its innovative recombination, suppression will not be complete and requirements for work will remain. Accordingly, an appropriate focus is on efforts to make the work easier to accomplish, requiring less effort to satisfy the requirements for work. A great deal of effort has been focused on this topic, especially in the field of software engineering which has created tools, principles and processes focused on reducing the difficulty of software work.

For example integration of code from two or 1000s of people is made more tractable by technologies from file diffs, to merge conflict reports in cvs, to git patch sets and github's pull requests and pull request discussion forums. Releasing is much improved by compiler

technologies such as universal binaries or build-systems, from make to Capistrano, that automate and regularize build, test and deploy, extending to efforts like the NMI Build and Test facility (Pavlo et al., 2006). Similarly user support can be facilitated by ticket tracking systems and customer relationship management systems. Even some aspects of synchronization work resulting from ecosystem complexity are the targets of technological time-savers, such as continuous integration extending beyond unit testing to integration testing in lead-user workflows (e.g., Trader, 2012). Some projects aim even higher, working to build infrastructure that automates software production itself, by mapping from mathematical proofs (e.g., Bientinesi et al., 2005) or machine learning techniques like genetic algorithms. These tools are akin to the application of capital machinery to improve the profitability of manufacturing, by both reducing costs and risks.

Design principles can also reduce the work needed. For example the principle of information hiding modularity is argued to reduce the complexity of integration work in production (Parnas et al., 1981). Ecosystem drivers of work can be reduced when producers follow the principle of only allowing slow change of interfaces for components on which much depends or other techniques designed to facilitate efficient evolution of software architectures (e.g., Garlan et al., 2009). Practice-led principles of collaborative development are also important, such as "avoid codebombs" or "head must always build" (Howison and Herbsleb, 2013) because they mitigate integration work, while governance principles such as Apache's action-oriented +1/-1 veto rules play a role (Fielding, 1999; e.g., O'Mahony and Ferraro, 2007). Other principles, such as the usefulness of cultivating a community of active users can reduce the impact of providing user support (e.g., Lakhani and von Hippel, 2003). Finally software process methods, such as agile methodologies are designed to reduce the gaps between requirements and development, and also make synchronization work easier, by bringing producers and potential users closer together through shorter cycles of development and release (e.g., Beck et al., 2001).

These technologies and principles can sharply reduce the amount of work necessary and are undoubtedly important to the overall challenge of sustaining scientifically useful software and software projects. Yet, just as the requirements for on-going work cannot be completely suppressed, two factors mitigate against efforts to reduce the difficulty of the work required as complete solutions to sustainability. Efforts in this direction never

entirely eliminate the need for work and they are themselves work to establish and sustain. Consider the initial effort needed in educating a lab in the proper use of git, or the effort to seek appropriate workflows for continuous integration testing. Even once established, techniques and technologies require effort to sustain them, such as the inevitable work needed in, for example, keeping a continuous integration system itself up to date.

In summary, then, tools and techniques are crucial to software sustainability and the cyberinfrastructure vision: without them the amount of work needed would simply be prohibitive. Yet work reduced by orders of magnitude is nonetheless work; if no one is available to do it then all the labor-saving technologies in the world will not sustain a project, nor the scientific usefulness of the software it produces.

3.3 Attract people willing to undertake the work needed

If the need for work to maintain the scientific usefulness of software cannot be suppressed nor effectively eliminated by making the work easier to do, then the work must be undertaken by people. This means that projects must attract human effort (and continue to attract it), drawing together motivated actors with appropriate skills to undertake work. The manner in which this is done we call the resource attraction system, which refers to collective mechanisms which establish incentives for people to participate in scientific software projects. We discuss three abstract resource attraction systems: commercial markets, open source peer production and scientific grant-making. While much could be said about each system, below we consider how they scale across the two dimensions of ecosystem context, particularly how they address complexity resulting from wider use-context diversity.

3.3.1 Commercial markets

A project selling software in a commercial market attracts resources by restricting the availability of its product to only those willing to pay, thus receiving revenue in the form of money. This money is then available to motivate work through the payment of wages or purchasing services from other market participants, thus motivating the accomplishment of the necessary work.

As the number of users rises, so does the revenue received as each user pays their licensing fee; resources available to the project rise linearly with the number of users. Assuming there are sufficient users willing to pay (a non-trivial condition) this enables the project to cover initial development costs and pay employees to ease use and distribution friction.

As the number of different use-contexts rises, projects employing commercial sales face the need for work driven by ecosystem context. Yet the act of a sale helps to accomplish insight into use-contexts, facilitating sensing and adjustment work. This is because as a side-effect of sales, a project drawing resources from commercial sales learns about their customers. Since a sale requires payment, companies learn the identities of their users, facilitating on-going contact. Sales themselves also give insight into the use-context of the customer. In small contexts this may result from detailed sales interactions as the company provides pre-sales support to users, work that is funded by the additional revenue derived from that specific sale. At larger scales, information about the suitability of the product in a changing software ecosystem comes directly from the pricing system, dissatisfaction reflected in customer’s declining willingness to pay.

3.3.2 Open Source Peer Production

Peer production is the resource attraction system that functions in successful non-commercial (or community-based) open source software projects (Benkler, 2002; von Hippel and von Krogh, 2003). Despite the common association, peer-production ought to be distinguished from "open source." Being open source is a characteristic of the code, while peer production is a characteristic of how it is produced: it is possible to be open source but not to be resourced by peer production (for example many grant-funded or even commercial projects are).

The literature on motivation to participate in open source has identified a set of non-monetary motivations, from the use value of the software itself, an opportunity to build reputation, an opportunity for learning, to a chance to express a communitarian ideology and to work in teams (Crowston et al., 2012; Roberts et al., 2006). Resources (in the form of direct labor) are attracted to projects that provide circumstances in which

these motivations can be satisfied (Benkler, 2002; Crowston et al., 2005; Howison and Crowston, 2014; Ke and Zhang, 2010; Michlmayr, 2003).

Despite the oft-celebrated differences from markets, the manner in which peer production attracts resources can be understood in a broadly similar fashion. The analogy to market allocation is clearest when considering the use value of software as a motivator: software that has use value (because it reduces a user's expenses) frees up money that can be directed to fund employee's participation in an open source project. Germonprez and Warner (2012) call this "leveraged development." The value generated by the use of the software is sufficient to motivate paying an employee to participate. But money doesn't have to be involved: a user that uses a piece of software to get their scientific work done might easily perceive that it is of value to them to do the work needed to include that component in their software assembly, or to undertake development work to improve a feature. In this sense resource allocation is decentralized and relatively undirected: participants build what they are motivated to build.

Unlike a market, however, the type of work that a prospective participant will do is linked to the motivations that attract them to the project in the first place (Conley, 2009; Dalle et al., 2009; Hertel, 2007; Howison and Crowston, 2014). Not all kinds of work can satisfy different kinds of motivations: a motivation to adapt the software to produce scientific plots for a particular scientific problem does not motivate the provision of support to other users.

As the number of users rises, peer production projects do not automatically gain additional resources, unlike commercial sales. In fact, peer production projects can be ambivalent to rising user numbers. Terry et al. (2010) found that developers saw non-contributing users as additional sources of user support burdens, rather than sources of rewards and motivations. On the other hand, developers motivated by reputation or status might see high users numbers as an advantage, although this is more likely to motivate development work than it is individual user support.

As the number of use-contexts rises, peer production faces intensified sources of complexity from ecosystem context. In fact the ease with which open source components can be combined implies more re-use of outside code than might be found in commercial sales, where each dependency might need to be separately licensed. Further, the freedom

to download open source code also implies that peer production projects do not have legitimacy to register their users and so do not have contact or tracking information for their users, reducing their ability to track change in use-contexts.

What peer production projects do have, however, is openness to contributions from their users, both in code contributions and by hosting discussion forums. Users are empowered and encouraged to alter component code, a characteristic open innovation researchers have called “actionable transparency” (Colfer and Baldwin, 2010). In this way peer production project users perform the sensing and often the adjustment work needed to deal with both exogenous and ecosystem context changes. The openness and ability to make changes to other people’s code creates the possibility of projects receiving information and partial solutions to changing usage contexts and collating them, passing solutions (or challenges) “upstream” to other projects. Finally, the emergence of software distributions, such as Debian, as independent peer production projects creates opportunities to manage the complexity of ecosystem context.

3.3.3 Grant-making

A third resource allocation system is particularly relevant in science: the provision of resources through grants provided by funding agencies, including government agencies and non-profit foundations. In the US, as one example, these agencies include the NSF, the Department of Energy and the National Institutes of Health, as well as foundations such as the Sloan Foundation.

In some sense grant money is akin to investment capital: it is made available with the hope of amplified future returns. Unlike venture capital, however, these anticipated returns are framed not as financial profit but in terms of achievement of more and better science. To this end agencies set aside a portion of their funds aimed at supporting science in general and choose to invest them in supporting software work relevant to science. In the words of the NSF’s implementation of the CIF21 Cyberinfrastructure vision, “Software is thus an integral enabler of computation, experiment and theory … [and] also directly responsible for increased scientific productivity and significant enhancement of researchers' capabilities” (NSF, 2012).

The particular investments made are guided by peer review and result in transfers of funds to projects which are converted to software work by providing rewarding opportunities for potential participants. This is particularly clear when projects pay directly for software work. Of similar importance are opportunities for activities resulting in scientifically valuable reputation, such as being among the authors of scientific papers.

Exactly how peer-review panels aimed at software work, in particular, choose what to fund is not well understood, but peer review panels in general emphasize scientific contribution, which has traditionally been closely linked to the production of knowledge instantiated in the scientific literature (i.e., publications), with an emphasis on both novelty and advancement of knowledge in the particular fields of the reviewers. Thus there are tensions between assessing what projects are likely to make contributions to, say, computer science and those likely to best facilitate science in other fields (e.g., Olson et al., 2008). This creates a tension between writing grants that promise novelty and transformation over needs to fund needed ongoing work.

Like peer production and unlike commercial sales, grant-funding is not directly linked to user numbers: as user numbers rise, support requirements rise but no additional resources are available. Larger user numbers are indeed important to component producing scientific software projects but only produce resources indirectly through future grant applications (Batcheller, 2011). In essence the project has to make a public-goods argument: the project is worthy of ongoing support because it provides a public good that would otherwise not be available, benefiting all members of the ecosystem. Projects have to periodically make the case that their continued contribution is sufficient to justify taking funds that would otherwise earmarked for direct funding of science. In other words, the project must argue that these funds ought to be, in effect, taxed at their source and given to the project to function as a service center, so that it can reduce the work that its users would otherwise have to do.

As the diversity of use-contexts rises, grant-funding offers no built-in mechanism to moderate the exponential growth of work driven by complex ecosystem contexts. Unlike projects using commercial sales, grant-funded projects do not attempt to control the distribution of their software; they do not have the prism of sales to provide insight into user assemblies. Yet unlike peer production, grant-funded projects are seeking to make the

argument that they reduce the work of their users and this complicates attempts to learn about use-contexts through openness to outside contributions. Rather, grant-funded projects must work directly to achieve a transfer of understanding about how the components are arranged into assemblies and how each of those assemblies is changing over time. These transfers take time, represent significant cost to grant-funded projects, and involve considerable interdisciplinary challenges as component producers seek to understand cutting edge science across a diverse range of use contexts (e.g., Faniel, 2009).

4 Policy Recommendations

Our analysis above has identified the management of complexity as key work that needs to be accomplished for sustainability in a software ecosystem. We also argued that of the available resource attraction systems, grant funding has the weakest mechanisms to either suppress, reduce or attract resources able to complete this work.

Science policy is limited in both legitimate goals and techniques. For example, the importance of preserving innovative freedom of action in science is paramount, rendering approaches that attempt to control end-users with binding detailed directions are unlikely to be considered legitimate. Similarly, attempts to reduce the exogenous factors driving work in the scientific ecosystem, the moving scientific frontier and novel technologies, are likely to be seen as counter-productive; putting the cart before the horse, as it were.

Nonetheless, two broad approaches are both legitimate and feasible. The first broad strategy is to improve the ability of the scientific software ecosystem to manage complexity by enhancing the grant making system itself: encouraging insight into end-user software assemblies, being welcoming to end-user contributions, and funding domain-specific distributions. The second broad strategy is to facilitate the transition of projects to alternative resource attraction systems, commercial sales and peer production, as appropriate.

4.1 Improve insight into scientific assemblies

A key challenge for policy to address is to enhance the visibility of the use contexts of scientific software components. In essence this means understanding how components

are arranged together to produce scientific results, enabling component producers (and others) to sense and rationalize the need for adjustment work at the edges. Visibility of end use contexts would also provide the possibility of anticipating changes in surrounding components and coordinating with other projects to minimize the need for adjustment work and contain potential cascades of reciprocal adjustment. Moreover, insight into usage can shape end-user behavior, driving coalescence to components through information cascades (Bikhchandani et al., 1992), as scientific end-users perceive what other scientific end-users are using and become preferentially more likely to use similar components. Insight into usage can therefore provide a lever to realize ecosystem architectures capable of suppressing complexity.

At present, however, component producers have surprisingly little insight into use, especially as it becomes widespread: they may know how many people have downloaded their software, (or even who has downloaded it, if they use a required sign up for download) and they may be able to search for citations to their software papers (assuming that they have made a clear citation request and that it has been followed by users). While these insights may help to demonstrate usage and scientific impact (and are indeed used by many projects, albeit imperfectly) they do not shed light on the complementary components and their dependencies. While projects may work closely with particular key users (Bietz et al., 2012, 2010) or convene domain-wide requirements gathering meetings, these methods are resource intensive and do not scale to the broad and deep insight required in situations of ecosystem contexts with high diversity of use-contexts.

Accordingly, science policy should focus on gathering and sharing insight into the software assemblies of scientific users. Happily this dovetails with the broad policy goal of increasing transparency towards reproducibility (e.g, Stodden et al., 2010). For example, contribution to code and data archives are being required at journals and conferences, leveraging a key influence point in the scientific world (Ince et al., 2012). Our analysis suggests an additional use of code archives: they can be aggregated and mined to understand complementarities and links between components, providing insight to component producers. A similar source of software assembly insight is available when scientific computing occurs in a cloud or distributed context, for example when projects access supercomputing resources. Supercomputing centers have focused on measuring

utilization of their computational resources, but not on gathering insight into the code running inside jobs. Nonetheless recent work has begun to record the use of libraries on remote scientific platforms, focused on optimizing utilization by ensuring that users are employing the best libraries (e.g., McLay and Cazes, 2012). Similarly, scientific gateways to which workflows (or, better yet, assemblies) are uploaded to be run or shared would be useful sources of data on software assemblies (e.g., Goecks et al., 2010; Roure et al., 2009; Stodden et al., 2012).

Focusing on obtaining insight into use contexts, rather than the hosting of reusable workflows or components, suggests alternative design emphases for these systems. For example, research would be needed to produce the most useful approaches (and therefore likely to be used by scientists) to creating such archives. Candidates include enhancements specifically aimed towards improving ecosystem insight, such as a step in the upload where components used are automatically detected and a selection interface presented to ask uploading users to confirm recognition of packages. A key issue here is to understand the legitimate privacy or competitive concerns of scientists and how to alleviate them, such as through trusted repositories, anonymization or sufficient aggregation.

Systems that actually enhance the existing workflows of scientists, rather than attempt wholesale changes in practice, might be more successful. One possibility would be analyzing the uploaded assemblies to help scientists identify which articles ought to be cited. Science policy-makers should encourage the creation of collections of software assembles and should fund research into mining collections of scientific software assemblies to explore appropriate techniques to identify sources of adjustment work, hidden complementarities and to best notify component producing projects.

A second approach to improving insight would be to incentivize funded projects to accept and cultivate contributions from end-users. As discussed above, peer production gains insight into use-contexts as users push their adjustments "upstream." Science policy-markers should make it clear that facilitating outside contributions (by those not supported by the project's grant) is appropriate and necessary for grant funded projects, and that having outside contributors does not mean that the project is having others do the work that the project was funded to do. Finally, policy-makers should make it clear that having outsiders contribute does not imply that projects are less in need of ongoing

support. Actions in this area can be as simple as asking funded projects to report on their efforts to attract outside contributors and highlighting success in creating contributing communities to peer-review panels as a positive towards continued or renewed funding. Establishing an online presence likely to encourage external contributions is an appropriate subject for project education, drawing on techniques from open source peer production where the goal is to establish "actionable transparency" (Colfer and Baldwin, 2010); the perception that outsiders can and should contribute.

A third approach is for science funders to incentivize synchronization work and the emergence of layered architectures capable of suppressing the need for synchronization work. While science policy-makers are limited in the directive power and legitimacy needed to enforce standardization in a top-down manner, insight can be drawn from the open source peer production world and encourage the emergence of distributions of software components. Distributions not only assist end-users by providing components in a form that eases the identification and location of dependencies, but they form a natural location for the coordination of software adjustments, both in pushing changes "upstream" to the most general component, but also in caching adjustments in time and suppressing costly circular cascades of adjustment work. Examples of funded distribution work are few and far between. The SBGrid project is a good example, focusing on providing coordinated software installs for Structural Biologists (Morin et al., 2013). Science policy-makers should issue specific solicitations for work to build domain-specific scientific software distributions, bridging between users and component projects and emphasize to peer review panels the complexity and importance of this work, both to the effectiveness of end-user scientists and to sustainability across the scientific software ecosystem.

4.2 Transitions between resource attraction systems

Science policy can encourage grant-funded projects to transition resource attraction systems, gaining both ongoing resources and the approaches to managing work derived from ecosystem complexity available to these systems.

Transitions to a model of commercial sales is familiar throughout science policy under the name of technology transfer. There are well-known examples of Scientific software that has made the transfer, especially in the statistical software space (e.g., SAS),

but many. Efforts in this direction include using cloud hosted services with a "freemium" model of free, broad, service provision for science and a paid tier for high and/or commercial users. Further research is necessary to undertaken contingencies to commercial sales as a sustainability approach but it seems clear that it is likely to only be successful in situations with high numbers of potential users but limited complexity in terms of use-context diversity (reaching a scale at which the price mechanism can communicate information about needs for adjustment) or in situations of low user numbers but those which have a capacity to pay substantial usage fees.

Transitions to open source peer production are often promoted as most appropriate for the sustainability of scientific software projects (e.g., Gambardella and Hall, 2006). Yet there is little understanding of how to build working peer production from grant-funded projects. Certainly simply making code available under an open source license is insufficient on its own to build a motivated and productive community, as shown by the predominance of individual and stalled projects in open source repositories (Krishnamurthy, 2002). Accordingly, science policy aiming at successful transitions must go further than requiring release under an open source license, which is the policy in place currently within some US federal agencies (e.g., NSF's SDCI and SI2 programs). An open source license is necessary but not in any sense sufficient. Transitioning from grant funding to open source peer production—or combining these models—implies substantial organizational change, including changes in team structure (from local to distributed), infrastructure (from controlled to open), governance (from hierarchical to shared), and commitment of participants (from predictable to unpredictable). Adding more difficulty still is the simple fact that a transition requires organizational change; it may in fact be far easier to achieve successful peer production from scratch than to begin with grant funding and then transition. Perhaps unsurprisingly, therefore, there are only a few examples of successful transitions from grant funding to sustainable peer production (e.g., Apache Airavata and the ENZO project). Therefore science policy makers should fund research on transitions, encourage projects which have accomplished successful transitions to share their approaches, and develop educational materials on guiding transitions. We need to understand what changes in project organization are needed and what actions projects can take to generate the needed changes. This knowledge would help peer-review panels

assess the quality of a grant applicant's sustainability plan and their plans towards transitioning from grants to peer production.

A variation of this approach would acknowledge that, as in entrepreneurial startups, the people best suited to beginning projects are not necessarily the best placed to continue and to build community around them. For the reason funding agencies could consider funding transitions from grant-funded development to peer production as separate projects, in essence seeking to seed "external" participation by funding it more directly, rather than supplying additional funds to the original developers.

5 Conclusion

We have argued that a primary driver of the challenge of sustainability in scientific software is complexity driven by high diversity of use. Sustainability is not a simple matter of improving the education of developers in techniques adopted from software engineering or commercial software development. Rather much of the complexity is driven by innovative recombination by scientists at the edge of their scientific frontier. Further we have argued that scientific grant-making, unlike commercial markets and open source peer production, currently lacks mechanisms to address this. We have identified a set of feasible and appropriate approaches, including further research, that scientific policy makers could take to improve the sustainability of the scientific usefulness of software.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grants SMA-1064209 and OCI-0943168. Versions of this work have been presented at departmental seminars at the University of Georgia and the University of Michigan. The authors thank Diane Bailey and Matt Germonprez for their insightful comments on earlier drafts.

References

Adner, R., Kapoor, R., 2010. Value creation in innovation ecosystems: how the structure of technological interdependence affects firm performance in new technology generations. Strategic Management Journal 31, 306–333.

Alexander, J., 2009. Software Sustainability through Investment.

Alt, F.L., 1964. The Standardization of Programming Languages, in: Proceedings of the 1964 19th ACM National Conference, ACM '64. ACM, New York, NY, USA, pp. 22.1–22.6.

Anderson, N.R., Ash, J.S., Tarczy-Hornoch, P., 2007. A qualitative study of the implementation of a bioinformatics tool in a biological research laboratory. International Journal of Medical Informatics 76, 821–828.

Atkins, D., 2003. Report of the National Science Foundation Blue-Ribbon Advisory Panel on Cyberinfrastructure.

Baldwin, C., Woodard, C., 2009. The architecture of platforms: a unified view.

Baldwin, C.Y., Clark, K.B., 2000. Design Rules: The Power of Modularity. Harvard Business School Press, Cambridge, MA.

Baldwin, C.Y., Clark, K.B., 2006. The Architecture of Participation: Does Code Architecture Mitigate Free Riding in the Open Source Development Model? Management Science 52, 1116–1127.

Batcheller, A.L., 2011. Requirements Engineering in Building Climate Science Software. (Ph.D. Dissertation). University of Michigan.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherl, J., Thomas, D., 2001. Manifesto for Agile Software Development.

Benkler, Y., 2002. Coase's Penguin, or, Linux and The Nature of the Firm. Yale Law Journal 112, 369–446.

Bientinesi, P., Gunnels, J.A., Myers, M.E., Quintana-Ortí, E.S., Geijn, R.A. van de, 2005. The science of deriving dense linear algebra algorithms. ACM Trans. Math. Softw. 31, 1–26.

Bietz, M.J., Baumer, E.P., Lee, C.P., 2010. Synergizing in Cyberinfrastructure Development. Comput. Supported Coop. Work 19, 245–281.

Bietz, M.J., Ferro, T., Lee, C.P., 2012. Sustaining the development of cyberinfrastructure: an organization adapting to change, in: Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work, CSCW '12. ACM, New York, NY, USA, pp. 901–910.

Bikhchandani, S., Hirshleifer, D., and Welch, I., 1992. Theory of Fads, Fashion, Custom, and Cultural Change as Informational Cascades. Journal of Political Economy 100, 1026.

Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R., Selby, R., 1995. Cost models for future software life cycle processes: COCOMO 2.0. Annals of software engineering 1, 57–94.

Boudreau, K., 2010. Open Platform Strategies and Innovation: Granting Access vs. Devolving Control. Management Science 56, 1872.

Brynjolfsson, E., 1993. The productivity paradox of information technology. Commun. ACM 36, 66–77.

Carver, J., Kendall, R., Squires, S., Post, D., 2007. Software Development Environments for Scientific and Engineering Software: A Series of Case Studies., in: Proc. ICSE. p. 559.
Colfer, L., Baldwin, C.Y., 2010. The Mirroring Hypothesis: Theory, Evidence and Exceptions (Working Paper No. 10-058). Harvard Business School Finance.
Conley, C.A., 2009. Work design for volunteers: The case of Open Source Software development, in: Best Paper Proceedings, Academy of Management Annual Meeting.
Crowston, K., Wei, K., Howison, J., Wiggins, A., 2012. Free (Libre) Open Source Software Development: What We Know and What We Do Not Know. ACM Computing Surveys 44, Article 7.
Crowston, K., Wei, K., Li, Q., Eseryel, U.Y., Howison, J., 2005. Coordination of Free/Libre Open source software development, in: ICIS 2005. Proceedings of International Conference on Information Systems 2005.
Dalle, J.-M., David, P.A., Rullani, F., 2009. Linking coordination, motivations and code structure in successful open source projects: A `stigmergic' approach, in: Academy of Management Presentation.
David, P.A., 1990. The Dynamo and the Computer: An Historical Perspective on the Modern Productivity Paradox. The American Economic Review 80, 355–361.
Dubois, P.F., 2005. Maintaining Correctness in Scientific Programs. Computing in Science and Engg. 7, IEEE Educational Activities Department–85.
Edwards, P.N., 2010. A vast machine computer models, climate data, and the politics of global warming. MIT Press, Cambridge, Mass.
Faniel, I., 2009. Unrealized Potential: The Socio-Technical Challenges of a Large Scale Cyberinfrastructure Initiative.
Fielding, R.T., 1999. Shared leadership in the Apache project. Association for Computing Machinery. Communications of the ACM 42.
Gambardella, A., Hall, B.H., 2006. Proprietary versus public domain licensing of software and research products. Research Policy 35, -892.
Garlan, D., Barnes, J.M., Schmerl, B., Celiku, O., 2009. Evolution styles: Foundations and tool support for software architecture evolution, in: Joint Working IEEE/IFIP Conference on Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Presented at the Joint Working IEEE/IFIP Conference on Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009, pp. 131–140.
Garlan, D., Perry, D.E., 1995. Introduction to the special issue on software architecture. IEEE Transactions on software engineering 21, 269–274.
Garlan, D., Shaw, M., 1993. An introduction to software architecture. Advances in software engineering and knowledge engineering 1, 1–40.
Gawer, A., Cusumano, M.A., 2002. Platform Leadership: How Intel, Microsoft, and Cisco Drive Industry Innovation.
Germonprez, M., Warner, B., 2012. Organizational Participation in Open Innovation Communities, in: Lundström, J.S.Z.E., Wiberg, M., Hrastinski, S., Edenius, M., Ågerfalk, P.J. (Eds.), Managing Open Innovation Technologies. Springer.
Gil, Y., Deelman, E., Ellisman, M., Fahringer, T., Fox, G., Gannon, D., Goble, C., Livny, M., Moreau, L., Myers, J., 2007. Examining the Challenges of Scientific Workflows. Computer 40, 32.

Goecks, J., Nekrutenko, A., Taylor, J., $author.lastName, $author firstName, 2010. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biology 11, R86.

Hannay, J.E., MacLeod, C., Singer, J., Langtangen, H.P., Pfahl, D., Wilson, G., 2009. How do scientists develop and use scientific software?, in: Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering. p. -8.

Hertel, G., 2007. Motivating job design as a factor in open source governance. Journal of Management and Governance 11, 129–137.

Howison, J., Crowston, K., 2014. Collaboration through open superposition: A theory of the open source way. MIS Quarterly 38, 29–50.

Howison, J., Herbsleb, J.D., 2011. Scientific software production: incentives and collaboration, in: Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW ’11. ACM, Hangzhou, China, pp. 513–522.

Howison, J., Herbsleb, J.D., 2013. Incentives and integration in scientific software production, in: Proceedings of the ACM Conference on Computer Supported Cooperative Work. San Antonio, TX, pp. 459–470.

Iansiti, M., Levien, R., 2004. The Keystone Advantage: What the New Dynamics of Business Ecosystems mean for Strategy, Innovation, and Sustainability.

Ince, D.C., Hatton, L., Graham-Cumming, J., 2012. The case for open computer programs. Nature 482, 485–488.

Jansen, S., Cusumano, M.A., Brinkkemper, S., 2013. Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry. Edward Elgar Publishing.

Ke, W., Zhang, P., 2010. Extrinsic Motivations in Open Source Software Development Efforts and The Moderating Effects of Satisfaction of Needs. Journal of the Association for Information Systems 11, Article 5.

Krishnamurthy, S., 2002. Cave or community: An empirical examination of 100 mature Open Source projects. First Monday 7.

Lakhani, K., von Hippel, E., 2003. How open source software works: “free”’ user-to-user assistance. Research Policy 32, 923–943.

Leyshon, A., 2001. Time-space (and digital) compression: software formats, musical networks, and the reorganisation of the music industry. Environment and Planning A 33, 49–78.

MacCormack, A., Rusnak, J., Baldwin, C.Y., 2006. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. Management Science 52, 1015–1030.

McCullough, B.D., McGeary, K.A., Harrison, T.D., 2006. Lessons from the JMCB Archive. Journal of Money, Credit, and Banking 38, 1107.

McLay, R., Cazes, J., 2012. Characterizing the Workload on Production HPC Clusters (Working Paper). Texas Advancd Computing Center.

Messerschmitt, D., Szyperski, C., 2005. Software ecosystem: understanding an indispensable technology and industry.

Michlmayr, M., 2003. Quality and the Reliance on Individuals in Free Software Projects, in: Proceedings of the ICSE 3rd Workshop on Open Source.

Morin, A., Eisenbraun, B., Key, J., Sanschagrin, P.C., Timony, M.A., Ottaviano, M., Sliz, P., 2013. Collaboration gets the most out of software. eLife 2.

NSF, 2012. A Vision and Strategy for Software for Science, Engineering, and Education: Cyberinfrastructure Framework for the 21st Century (CIF21) (Dear Colleague Letter No. 12-113). The US National Science Foundation.
NSF Cyberinfrastructure Council, 2007. Cyberinfrastructure Vision for 21st Century Discovery.
O'Mahony, S., Ferraro, F., 2007. Governance in Collective Production Communities. Academy of Management Journal 50, 1079–1106.
Olson, J.S., Hoder, E.C., Bos, N., Zimmerman, A., Olson, G.M., Cooney, D., Faniel, I., 2008. A Theory of Remote Scientific Collaboration.
Parnas, D.L., Clements, P.C., Weiss, D.M., 1981. The modular structure of complex systems. IEEE Transactions on Software Engineering 11, 259–266.
Pavlo, A., Couvares, P., Gietzel, R., Karp, A., Alderman, I.D., Livny, M., Bacon, C., 2006. The NMI build & test laboratory: Continuous integration framework for distributed computing software, in: The 20th USENIX Large Installation System Administration Conference (LISA).
Reay, D.S., 2010. Lessons from climategate. Nature 467, 157–157.
Roberts, J.A., Hann, I.-H., Slaughter, S.A., 2006. Understanding the Motivations, Participation, and Performance of Open Source Software Developers: A Longitudinal Study of the Apache Projects. Management Science 52, 999.
Roure, D.D., Goble, C., Aleksejevs, S., Bechhofer, S., Bhagat, J., Cruickshank, D., Fisher, P., Hull, D., Michaelides, D., Newman, D., Procter, R., Lin, Y., Poschen, M., 2009. Towards Open Science: The myExperiment approach. Concurrency and Computation: Practice and Experience John Wiley \& Sons, Ltd.
Ryghaug, M., Skjølsvold, T.M., 2010. The global warming of climate science: Climategate and the construction of scientific facts. International Studies in the Philosophy of Science 24, 287–307.
Segal, J., Morris, C., 2008. Developing Scientific Software. IEEE Software 25, 20.
Stewart, C.A., Almes, G.T., Wheeler, B.C. (Eds.), 2010. NSF Cyberinfrastructure Software Sustainability and Reusability Workshop Report.
Stodden, V., Donoho,, D., Fomel, S., Friedlander, M., Gerstein, M., LeVeque, R., Mitchell, I., Ouellette, L.L., Wiggins, C., 2010. Reproducible Research. Comput. Sci. Eng. 12, 8–13.
Stodden, V., Hurlin, C., Perignon, C., 2012. RunMyCode.org: A novel dissemination and collaboration platform for executing published computational results, in: 2012 IEEE 8th International Conference on E-Science (e-Science). Presented at the 2012 IEEE 8th International Conference on E-Science (e-Science), pp. 1–8.
Terry, M., Kay, M., Lafreniere, B., 2010. Perceptions and practices of usability in the free/open source software (FoSS) community, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '10. ACM, New York, NY, USA, pp. 999–1008.
Trader, T., 2012. Blue Waters Opts Out of TOP500. HPCWire.
Trist, E.L., Bamforth, K.W., 1951. Social and psychological consequences of the longwall method of coal-getting. Human Relations 4, 3–38.
Von Hippel, E., von Krogh, G., 2003. Open Source Software and the `Private-Collective' Innovation Model: Issues for Organization Science. Organization Science 14, 209–223.

West, J., 2006. Scope and Timing of Deployment: Moderators of Organizational Adoption of the Linux Server Platform. International Journal of IT Standards Research 4, 1–23.
Yegge, S., 2011. Stevey's Google Platforms Rant I was at Amazon for about… [WWW Document]. URL https://plus.google.com/+RipRowan/posts/eVeouesvaVX (accessed 12.24.13).