1

Practicing Open Science

Luis Ibanez

Kitware, Inc.

William J. Schroeder

Kitware, Inc.

Marcus D. Hanwell

Kitware, Inc.

CONTENTS

1.1 Introduction 2
1.1.1 The Evolution of Scientific Community 2
1.1.2 Sharing Is Essential 3
1.1.3 Reproducibility and Open Source 4
1.1.4 In Pursuit of Open Science 5
1.1.5 Organization 5
1.2 Open Data 6
1.2.1 Plan to Share Data 6
1.2.2 Data-Centric Computing 7
1.3 Open Source 7
1.3.1 Version Control and Provenance 8
1.3.2 Automated Testing 9
1.3.3 Unit Testing 11
1.3.4 Code Review 11
1.4 Open Access 13
1.4.1 Open Access Journals 13
1.4.2 Versioning Documents 13
1.4.3 Open Reviews 14
1.5 Open Standards 14
1.6 Open Science Platform 15
1.6.1 Midas Platform 15
1.6.2 CMake-Based Software Process 16
1.6.2.1 Overview 17
1.6.2.2 Unit Testing 18
1.6.2.3 Examples of Code Review 20
1.6.3 The Insight Journal 21
1.6.3.1 Practical Details 22
1.6.3.2 Community Involvement 23
1.6.3.3 Data Concerns 23
1.6.4 Scalable Computing 24
1.6.4.1 High Performance Computing 25

1.6.4.2 Science as a Service 26
1.7 Challenges 28

1.1 Introduction

Science is a system for gathering knowledge, and developing explanations and predictions about the universe in which we live. A central tenet of this system is that something is known only when multiple, independent observers agree on a common experience. That is, experiences (which are more commonly called experiments) are reproducible. As such, the scientific method is by definition open; it is only when independent parties precisely replicate an experience that experimental results are considered valid.

Despite this basic, and obvious tenet of openness, the pressure to close science is growing. Due to the exceptional innovative power of science, commercial interests, and personal career pursuits, many scientists and research institutions are heading down the path of secrecy and strong protective measures. In addition, publishers which garner financial benefit from controlling the dissemination of scientific knowledge, have been reluctant to openly share information, and are under increasing financial pressure to protect what they consider their intellectual property. Consequently, one of the tragedies of the current era is that the term "Open" must be prepended to Science despite the fact that openness is a fundamental requirement of the scientific method. Open Science is a redundant, descriptive phrase, yet it has become necessary to remind ourselves that we must maintain openness if we are to effectively practice science.

Countering the pressure to close science is of course the emergence of the Internet. This ongoing Web revolution has given rise to near zero-cost methods of disseminating information, meaning that the ability to share the results of scientific research has been greatly enhanced. It is more than a simple matter of playing nice and sharing with others, the increasing complexity of modern science demands sharing and collaboration, since large teams with multidisciplinary expertise are required to address current challenges and gather advanced knowledge (see Section 1.1.2). Thus it may be that scientific progress will stall without greater openness, and scientists will have no choice but to share, and share more effectively.

The conflict between sharing and secrecy has been present since the earliest days of scientific practice. Some scientists have routinely hidden or encoded their data, and often released it only as necessary to support their work, or once personal career achievement was assured. However it is clear that the practice of science is changing rapidly, with key players such as publishers and societies, as well as scientists themselves, under significant pressure to change their ways. Thus the conflict is taking on deeper meaning, and is nothing less than a revolutionary reevaluation of how we practice science.

1.1.1 The Evolution of Scientific Community

As the scientific method was developed, the demands of reproducibility, and hence the need to share information, quickly gave rise to scientific societies and publications. For example the Royal Society was established in 1660 with the enviable motto (translated from Latin) "*Take nobody's word for it.*" Effectively the role that this society and the many following it took was one of *community building*. Early on, meetings were held in which experiments were jointly performed (the earliest form of peer-review), and eventually results were codified, published, and shared. Given the technology of the time, this process rapidly evolved into a journal-based system in which communications between scientists were collected and distributed (for a fee) to subscribing recipients (see Figure 1.1).

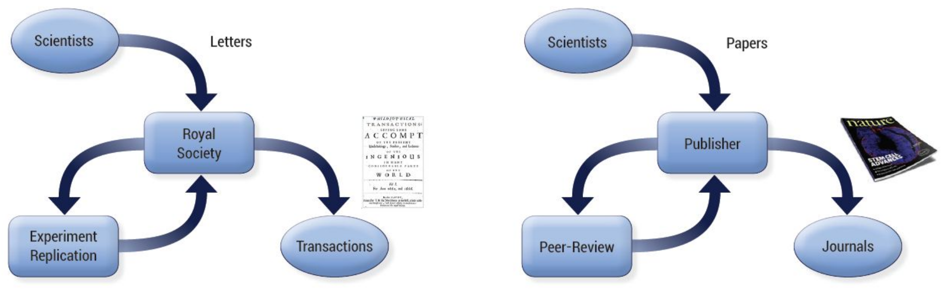


FIGURE 1.1: The evolution of the practice of the scientific method, from a society-oriented process of reviewing, and verifying reproducibility (left) to today's peer review based publishing process (right). Without openness, the peer review process cannot verify reproducibility.

Fast forwarding to the current day, scientific publishing has grown into a multi-billion dollar industry. It has served science well over the centuries by gathering information and sustaining scientific communities, including sponsoring conferences and facilitating the peer review process. However, the Internet has unleashed new ways to grow and support communities, as a result the publishers are feeling the inevitable financial pressures and demands for process change. The current, ponderous model of peer review and paper-centric publication has been viewed by many, as far too slow, and limited in the amount of information that is conveyed. The main deficiency being that journal articles do not always provide the information necessary to reproduce a result.

The notion of what is a scientific community is also changing rapidly. Conventional publishers might choose to maintain their old ways (and profit margins) if they had a choice, but with the low cost of exchanging information and serving communities it is clear that change will continue unabated. In the end the publishers will only survive by returning to their roots: serving the scientific community. This may mean taking a supportive role by adopting new methods for curating, organizing, and coordinating scientific knowledge, as well as continuing the support of communities through various interaction forums. In particular, conferences, data hosting and repositories, to name a few. In the mean time, many scientists and institutions are taking matters into their own hands and using reproducible methods such as those described in this chapter to further the reach of science.

1.1.2 Sharing Is Essential

Scientists are taught that the scientific method rests on three pillars of practice: experimentation, theory, and computation. Recently a fourth pillar has been proposed, data-intensive computing [25], although many consider it part of the computational pillar. Until recently, the standard publishing model that uses written articles to describe experimental apparatus, articulate theory, and codify computation was reasonably sufficient to support reproducibility and therefore scientific progress. Unfortunately this model is no longer adequate: the complexity of experiment, theory and computation is such that a brief paper cannot even begin to capture the detail necessary to describe an experiment to the point where it could be reproduced by a third party.

Consider a paper in computer science, an area in which the authors are familiar. A typical eight-page paper, or even an extended paper, can never describe the nuance behind complex algorithms. For example, an advanced algorithm may require many dozens of control parameters, not to mention internal data structures that can greatly affect performance and accuracy. In our experience we have found that actually reproducing such an algorithm may require years of effort, and in doing so we invariably fall back on the help of the original author who sheds light on "implementation details" which are frequently omitted in journal articles. The time demands to verify reproducibility are so large, that if we as authors were to reimplement algorithms for research purposes it is unlikely that we would find the time to develop our own line of research. As a result, many experiments are never reproduced (especially in peer-reviewed documents), with the additional burden on researchers who spend inordinate time reimplementing what has been done before, due to lack of access to the original software implementation of published works. Thus without the necessary sharing, the scientific endeavor is choking on complexity and resting on unstable foundations.

There are further, selfish motivations to practice openness: there is correlative evidence supporting the notion that: sharing furthers a scientific career. Recently [29] posted an article that suggests that sharing materials results in greater citation of the published material. Further, some argue persuasively that collaborative teams who, by definition, share information are the future of science [8].

1.1.3 Reproducibility and Open Source

The goal of this chapter is quite practical: to share some of the methods that we use in our practice of science to ensure reproducibility and encourage community building. To that end, we have learned much from our participation in various open-source projects, of which we are all developers and contributors. Indeed, the principles and practices of scientific reproducibility have been imprinted within the DNA of the Open Source movement since its inception. This is no coincidence, it is the consequence of the fact that the Open Source movement originated in an academic environment and more specifically it was kindled in research laboratories. In Open Source communities, reproducibility is ensured through the practices of code review, unit testing, continuous integration, public documentation, open mailing lists and forums. In this chapter, we show how these methods, and extensions to them, can be used in the practice of science.

In general, open source communities are far ahead of most scientific communities when it comes to the practice of reproducibility. In large part this is due to the rapid evolution of the open source movement in response to the growth of the Internet and the Web. In the meantime, most of the scientific community has remained constrained by the limitations of a process entrenched in practices that date back to the introduction of the printing press.

One of the goals of this chapter is to describe how the practices of Open Source communities are being brought back to mainstream scientific research. This is based on many years of work developing open source software for scientific applications. During this time we have regularly interacted with scientific research teams, providing software engineering support for them, bringing their algorithmic implementations up to the standards expected in modern software engineering, particularly with regard to testing, and facilitated the reuse of their software and data, through widespread sharing of resources.

These practical experiences are presented here with the aim of encouraging the scientific community to adopt them in their daily work. Such adoption typically involves working at several levels simultaneously. We have found that it is important to work in parallel at the following levels:

- Cultural,
- Rewards/recognition,

- Career building,
- Funding, and
- Technical training.

A common hurdle to adopting open source principles is the confusion that individuals and organizations face when classifying their challenges. For example, it is common to find that a technical difficulty can be misinterpreted as a cultural challenge, or that a disincentive in the funding space is mischaracterized as a technical problem. As we go through the topics in this chapter we will attempt to properly identify the challenges and opportunities in the adoption of reproducibility verification, and how they relate to the specific levels listed above.

1.1.4 In Pursuit of Open Science

The Open Science movement is at its core an attempt to correct behaviors in the scientific community and return to an environment where reproducibility is again at the center of scientific research activities. Practicing Open Science requires four fundamental ingredients:

- Open data,
- Open source,
- Open access, and
- Open standards.

Each one of these ingredients is necessary to realize the core aims of sharing results and stimulating scientific progress. Open data provides the opportunity for verification, analysis, and subsequent publication of scientific results in new forms. Open source embodies scientific methods, so that new computational processes can be independently examined, reviewed and reused. Open access facilitates the review and validation of research processes and results. Finally, open standards, while not absolutely essential to Open Science, simplify the process of exchanging data, methods and publications thereby accelerating the research process.

We see a broader role for Open Science and its impact on society. The Web is opening up new lines of communication, providing access to scientific results which can be viewed by virtually anyone with an Internet connection. This includes the general public who often have strong interest in scientific research, for example when learning about current medical treatments. However current practices such as pay walls and overly aggressive data rights limitations are impeding realization of its full potential. The current situation requiring authors to sign away copyrights to publishers, despite the fact that results are often produced with the support of public funding, simultaneously represents an impediment to progress as well as a significant public subsidy to narrow business interests. Instead, permissive data rights using open licenses such as CC0, or CC-BY, are necessary to return the spirit of community to science, and ensure its role as an effective driver of innovation and major contributor to societal progress.

1.1.5 Organization

This chapter is organized into two major parts. With the introduction behind us, the first part consists of four sections discussing in general terms issues relating to open data, open source, open access, and open standards. Next, the second part (in Section 1.6 *Open Science Platform* elucidates specific tools and practices that the authors use in their daily practice of science. Naturally this section has a strong software orientation as the authors are computational scientists (of sorts) by training. However as science is becoming ever more computationally driven, we hope this material will be of interest to scientists of all persuasions. Finally, we conclude with a brief section on the future challenges of *Practicing Open Science.*

1.2 Open Data

To a significant extent, the scientific methods concerns itself with gathering, analyzing and deriving data, partially to perform the essential work of acquiring knowledge, but also to buttress explanations and support predictions. Data plays different roles in each of the three scientific pillars of experiment, theory and computation; and naturally supports each during the process of scientific investigation. For example, gathering data through experiments or direct measurement is necessary to subsequent data analysis, typically to develop theories of causality and correlation. On the other hand, theories are used to inform computation, which generates predictive output data, which is typically compared to experimental results to *falsify* [34] theories and refine computational models. *Thus data serves as the focal point in the scientific workflow, and unfettered access to it is required for the scientific process to proceed efficiently.* Without such open access to data, the power of science to produce knowledge, and thereby drive innovation and economic progress, is severely impeded.

Despite the obvious necessity of unfettered access to data in order to support the scientific process, there are several powerful forces that create barriers. Many scientists place career goals above sharing, as a valuable data set may generate an important body of work and hence citations. Some scientists also insist on withholding data (at least for a short period) while they verify its correctness (and hence preserve their reputation). While these reasons certainly have merit in the real world, it is easy for them to become unbalanced behaviors that significantly impede scientific progress, or in the worst case violate the core principle of reproducibility.

Finally, commercial interests represent another set of growing pressures to withhold data or impede its reuse: data can be withheld because it is deemed valuable, or copyright may be used to limit its distribution. What is unfortunate about these barriers is that, not only do they interfere with potentially specialized lines of research, they all but prevent large-scale meta analysis across potentially thousands of data sets. For example, consider the case of automated access followed by statistical analysis over dozens of disparate data sets from a variety of sources. Having to formally request access to data one instance at a time is not feasible; automated meta-analysis depends on ready access, and to a lesser extent, open APIs (see Section 1.5). Furthermore, restrictive licensing can prevent derived data sets from being published (or severely limit the breadth of the source data).

1.2.1 Plan to Share Data

Until recently, the central role of data was implicit to the scientific process. Data was modest in size and could be exchanged among the community through tabulation of published results. Then, in the very recent past, data grew significantly in size. This has led to significantly more complex data sharing, requiring computers and associated storage media such as tapes, floppy disks and hard drives to exchange information. At about this time scientific publications began referring to external data sets; and now with the advent of the Internet these data could be placed on a public site and distributed across the scientific community. While this process continues today, the sharing process implicitly depends on the scale of the data. Modest-sized data can be exchanged and if necessary, reacquired (if scientists decide not to share it) at reasonable cost. However, this process is changing rapidly as data size increases—it is becoming increasingly hard to exchange large data, and the re-acquisition of data is often prohibitively expensive.

Consider the current state of affairs. The cost to acquire data, or generate (simulate/calculate) data on a supercomputer can be extremely expensive. For example the original cost to sequence the human genome was nearly one billion dollars. The size of data is growing rapidly too, with terabyte data sets becoming common, with petabyte data sets emerging (and exascale is on the horizon). At this scale, data is too large to easily exchange (the copy operation can take weeks or months even with high bandwidth links); and too expensive to reacquire. Thus the costs and practical data transfer considerations are driving science in a direction that absolutely demands better sharing of data. In the past, data was exchanged by researchers who felt the ethical obligation to share information for the purposes of advancing science. Unfortunately data was occasionally withheld, or publication delayed, for the purposes of validation, or worse yet, due to competitive career motivations. On a small scale this had modest impact on the the advancement of science; however, at the current scale, sharing data has become vital to scientific progress. In our opinion, it is imperative that scientists include data sharing plans as part of future funding proposals; indeed many US Federal organizations have put in place requirements for such plans. This is the case for The National Institutes of Health [12] for example.

1.2.2 Data-Centric Computing

The practical concerns related to the cost, size and scale of "Big Data", combined with the philosophical motivations to publish scientific materials across the larger community, have led to new models of data distribution and curation. Data-Centric Computing [25] is one such response. In this approach, data is central to the workflow Figure 1.2; once acquired, generated, or computed data is left in place in a "central" repository (in practice the data repository can be distributed across the web depending on where it is acquired or computed). Access to the data is enabled through the Web. Client-server architectures are employed wherein the server resides directly alongside the data, and clients are used to access, analyze, visualize, organize and otherwise curate data. In addition, it is expected that extended research teams are working together on the data, meaning that simultaneous data access, and collaboration must be supported.

The point here is that these data repositories represent significant scientific resources, and the work flow inevitably revolves around them. The scale of the data is so vast that multiple collaborative teams are required to ferret out useful information. With the expense and complexity of data, these data must be recognized as resources that are readily shared, and are accessed through means of open standards and programming models.

1.3 Open Source

Much of today's science depends on computation, which to ensure reproducibility must be completely defined. Due to the complexity of computational methods, it is no longer possible for brief descriptions or pseudo-code in a publication to properly characterize methods. The

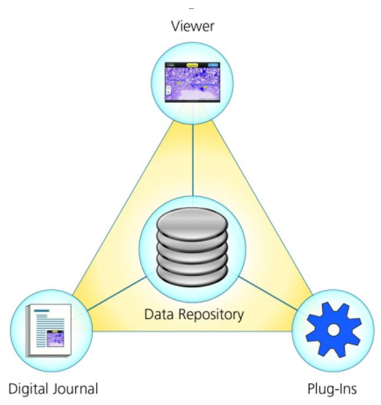


FIGURE 1.2: Data-driven scientific process. Data, once acquired or generated, is rarely moved. Rather visualization, analysis, and journaling process orbit the data repository. This requires special architectures (e.g., client-server) and to locate the computational resources close to the data.

demands of reproducibility, and hence Open Science require full disclosure, which means providing source code, execution parameters, and the computational environment—everything required to exactly reproduce an experiment.

A large portion of the practice of Open Science has been informed by the Open Source movement. In this section we describe the various methods used by open source communities to ensure reproducibility. In particular we focus on access to software through version control systems; automated testing to ensure reproducibility; and systems for code review to validate the correctness of the software.

1.3.1 Version Control and Provenance

Carefully tracking changes to scientific data, methods and publications is essential to the scientific process, especially as part of the verification of reproducibility. This curation of scientific resources is also fundamental for educating future scientists, who will have the opportunity to inspect in great detail how particular experiments were performed in the past. In the open source world, the ability to track changes is referred to as *version* or *revision control*.

Version control has been at the heart of the open source software movement since its earliest days. Initially version control was crudely implemented as a collection of tools to create patches on top of the original files, email them and apply the patches on the receiving end. Source code was freely available, approaches could be discussed and changes proposed via these patches, clearly indicating changes made to the original. As this practice became widespread, more sophisticated tools were developed, but at their core they were designed around moving patches more efficiently over the available transport options. Today we are fortunate to have a wide array of sophisticated version control systems available, with many powerful and open source systems under active development.

One of the most popular version control systems is Git, a project initiated by Linus Torvalds to manage the flow of patches produced by Linux Kernel developers. It is perhaps one of the most sophisticated, offering a vast array of options that can be daunting for novices. It is part of a new generation of version control systems, called *distributed version control systems* (DVCS). The major improvement over previous systems is that users can “clone” a repository to receive a full copy of the source code and all of the changes that were ever applied to it. This is in stark contrast to many alternative, centralized version control systems such as RCS, CVS and Subversion in which developers receive only the tip of the current branch and only the central repository server contains the complete history of the project.

There are many advantages to DVCS, one of the most important being the ability to easily create a local directory which can then be initialized and placed under version control, and then readily shared with others. New files can be locally committed and their contents stored. Later, when changes are made, these can be recorded in the form of patches applied to the original, giving the author total freedom to look at all previous versions of a given file. If the project grows, then sharing the project along with its history is simple, whereas centralized version control systems require up front planning and coordination with the central repository maintainer. Another important advantage to DVCS is that the source and all history can easily be mirrored at multiple locations, with private branches that can later be published in public repositories for all to see. This allows for work to take place in private when necessary, that can later be shared with full history.

Another often overlooked, but powerful capability of version control systems is the ability to track code provenance. Not only are the files and all modifications stored, the date and time of each modification is stored along with the author and a message detailing the reason(s) for the change. This tracking is implemented in the form of *commits*, which mark events in which a particular set of changes were applied to the content of the repository. Particular points in history can be *tagged* to indicate major events such as software releases, and signed using encrypted keys to assure that a particular tag was signed by a given person using a cryptographically secure signature. Due to the nature of systems such as Git which use special hashes to establish the identity of a commit, it is possible to detect alterations to previous commits that the signed commit depends upon, thus offering high levels of data integrity. It is not necessary to compare all files against a known good copy, just the hash of the commits you have to a signed copy that you trust. This offers a desirable degree of data provenance, using openly verified algorithms for establishing data integrity, that is difficult to obtain with other approaches.

1.3.2 Automated Testing

Moment of Zen:
What scientists call: *Experiments*
Open source developers call: *Tests*

In this subsection, we equate the scientific concept of *performing an experiment* with the open source practice of *running a test.* In today's world of scientific computational research, these two actions are one and the same.

The scientific principle of *verification of reproducibility* is implemented in open source communities by relying on automated processes. The reason is simply that software systems have a natural tendency to develop into large and complex systems. In such an environment, the informal notion that: *We attempt to replicate today an experiment that we did yesterday*, can not be left to the fallibility of good intentions, it must be formalized.

The bottom line is that we are forced by practical necessity to script automated processes that can be run repeatedly. This is because the accumulation of *the things that we did yesterday*, and the ones done the *day before*, and the *day before that one*, rapidly become a combination of thousands of experiments. Attempting to repeat them by manual execution guided with notes or plain memory, simply does not scale and discourages practitioners from actually running all the experiments.

Automation not only makes reproducibility practical, it also makes it reliable. By capturing the process describing how to repeat an experiment, automation forces the practitioner to script every single detail that is relevant to its execution. This means leaving nothing to informal processes, local idiosyncrasies, or good intentions.

The practical way to encourage developers and researchers to automate their tests is to ask them to run them on a **daily basis**. With a set of even tens of experiments that must be run every day, a methodology to automatically run these experiments, emerges quickly. This is an example of how a **cultural** requirement leads to **technical adoption**. Unfortunately, the converse is also true. That is, in a laboratory environment that does not automate its experiments, staff quickly grow accustomed to not running the experiments on a regular basis, reinforced by the excuse that it will simply take too long to do it. While there are other more urgent tasks to tend to automated tests will be neglected. This chicken and egg problem can be solved by working first at the cultural level and developing a sense of reputation and pride in the craftsmanship of being *the one who runs their experiments daily*.

Developers reputations are built in open source communities through practices of transparency, peer-review, and accountability in a meritocratic process.

Transparency is achieved by publishing, on public web sites, daily test results. It quickly becomes obvious who does and who does not run tests on a regular basis.

Peer-review is performed by the larger developer community, who routinely scan test results as part of their daily software development work. The more formal practice of *code review*, drills down into the changes that another developer may have made to the system (a prerequisite of code review is that all tests are run). During this exercise, it is easy to expose whether the original developers actually ran the tests before and after making changes. When a reviewer finds that a developer failed to run the tests, it is culturally expected that a public admonishment is in order. This is typically done in a cordial way, and sometimes with a humorous tone. The intention is not to provoke a confrontation, but to enforce a social norm. Not running tests in an open source community is simply *bad etiquette*. It is frowned upon, the same way as if you were to sneeze on a colleague's sandwich.

Accountability is a follow to the transparency and peer-review practices in open source communities. It comes down to the implicit rule that: *if you broke it, you fix it*, as a way of redeeming your reputation, with the caveat that if someone else fixes it for you first, then your reputation is damaged and the reputation of the person who fixed it is enhanced.

The combination of social, cultural and technical practices build an environment in which to be a good member of the community, tests are diligently maintained and run frequently. As a consequence, the testing process is automated in such a way that they can be run with minimal effort.

The notion that a *gradual* system of sanctions must be implemented in order to enforce compliance with community-established rules is one of the elements that Elionor Ostrom (Nobel Laureate in Economics 2009) identified as a result of studying self-governing communities who manage common resources in fields as diverse as fisheries, underground water basins, forests, and irrigation systems. Her contention is that, a gradual system of sanctions is essential for the successful self-governance of the Commons, in the absence of government intervention or the use of property systems [32]. These are indeed the conditions under which both open source communities and scientific communities operate on a regular basis.

1.3.3 Unit Testing

Unit testing is the translation of the principle of Occam's Razor to the daily practice of software development. In particular, it is the quest for the minimal explanation for a given behavior. The goal of unit testing is to empower developers to rapidly pinpoint the root cause of problems in the software. In particular, it is important to not rely on complex tests that involve the execution of thousands of sections of the software project. Otherwise, when a complex test fails, it is extremely difficult to figure out the root causes of the failure.

Unit testing takes the approach of verifying the correctness of the most basic components of the system, and in the process, to build confidence in the behavior of each component to the point where it is possible to rapidly locate which one of the many pieces of a software package is causing a problem.

Unit testing is not just a software practice—it is *a state of mind*. The practice is motivated by the same principles at the core of the quest for reproducibility verification in the domain of scientific research: *Acceptance of the fact that errors are ubiquitous*. Therefore, the only way to keep errors at bay is to continuously set traps for them at every corner of every experiment. The presumption is that *errors are indeed present*, and therefore it is important to put in place tests that check for the presence of errors at every point in the process. It is the *continuous failure to find errors*, combined with the thoroughness of the testing efforts, that builds confidence in the correctness of the overall process.

The daily practice of unit testing also leads to the principle of decomposition, by which complex problems are partitioned into smaller units, and then those units are implemented and tested independently. This practice leads to better designed software, which is clearly organized and easy to maintain. Requiring unit testing as a cultural practice forces developers to stay away from building large and complex pieces of monolithic software, and to instead modularize their designs and build more general, robust and reusable components.

Practitioners who employ unit testing write the test at the same time they write the code, in a rapid iterative process. They start with an empty piece of code, and then write a test for the first minimal feature. The tests will at first fail, given that the feature is not yet implemented. The developer will then implement the feature, and bring it to the level where it passes the test. Note that it is important to ensure that the test fails prior to implementing the feature, thereby validating the test itself.

The successful practice of unit testing is closely tied to the application of agile methodologies in software development (see Section 1.6.2. The practice of unit testing requires that one writes features in small incremental cycles; designing, implementing, testing the code, and then iterating back to revisit the design.

This way of working is conceptually no different from what any experimental researcher should do on a regular basis. For example, checking that the chemical reactions that they are about to use are pure enough, verifying that the thermometer to be used in an experiment is actually in a working state and correctly calibrated; and overall, ensuring that the experiment is performed in a controlled environment with as few uncertainties and systematic errors as possible.

1.3.4 Code Review

Code review is a fundamental practice of quality control in which developers review the changes made by their peers, in the quest to spot potential errors, unnecessary features, and ensure consistency with the overall design and style of a project. There are different methodologies for implementing code reviews, but many elements remain common.

One form of code review, that is used in many software projects, unfolds by having developers perform reviews in an ad-hoc fashion with heavy reliance on the version control system. Review often take place after changes are merged with the main code base, producing a stream of revision control commits. Developers subscribe to a mailing list that sends an email with the contents of each commit made to the repository. Developers take time to read through these commits, checking those relevant to them and either fixing any problems they notice, or emailing reviews to the relevant development list (or via private email channels). This form of code review is quite common in projects using centralized version control systems for those developers who have commit rights.

Another form of review that has been employed for decades is the practice of emailing patches to a development mailing list for review. Developers then respond with high level reviews and/or line-by-line comments, and then iterate and modify the patch until it is deemed ready for commit. This practice not only serves to foster higher code quality, but it educates new developers in the expected code style, pitfalls and common practice of the software project. Several variations of this basic procedure include attaching patches to bug reports, performing review in the bug tracking system, or using dedicated code review platforms where patches can be uploaded.

With DVCSes an alternative model has emerged. Distributed version control enables a developer to develop in a new, private development branch, and to apply a sequence of changes to that branch in the form of commits. Developers can create as many of these branches as they wish. Given that every branch contains an independent history of the project this mechanism enables developers to undertake modifications to the project without interfering with the work of other developers, yet with the ability to share their work with any of their peers. The developer can also push branches of their choosing to multiple remote locations.

Software tools for code review, such as Gerrit, support remote repository locations where branches are pushed with proposed changes. These changes can then be displayed in a web application for the entire community to see, with an associated set of access control lists specifying permissions for developers of the project. Developers are then able to work freely on their code as they normally would, and when the code is ready to be merged into the main code base, or reviewed by a wider audience, it is pushed to Gerrit. Once pushed, Gerrit reviewers can be assigned to a topic, and the system will notify them. They can then make general comments about a commit, or comment on particular lines with questions or comments. These comments are seen by all users of the system, along with the author of the topic. The author can then respond to the review, possibly uploading edited versions of their commits, until the code is approved.

Once code is approved it is given a score indicating approval, this is also recorded using a mechanism recently added to Git called notes and uploaded along with the changes when they are merged. This creates a permanent record of who reviewed the changes along with links back to the review. If bugs are later found it is possible to go back to the original review if more detail is desired beyond what was recorded in the version control system. The code review process can also be significantly enhanced using various automated build, test and analysis techniques (such as those described later based in Section 1.6.2). Pre-testing before committing to the main branch enables developers to assess proposed changes before inclusion into the system proper.

The use of code review can seem like an unnecessary drain on resources, but it is usually much cheaper to review and catch mistakes before they are merged than to track them down afterwards. If good tests are written, and careful code review is performed, it is much easier to bring new developers into a project and empower them to make significant changes with less concern for inadvertently breaking the system. Often new developers fail to adhere to established practices which if caught in an initial code review can be corrected very early on. If such problems are missed, weeks or even months of development effort may pass before the errors are detected and fixed, with the added cost that a good deal of the development that happened in the meantime will also have to be corrected to conform with expected standards.

1.4 Open Access

The public dissemination of scientific knowledge is essential to promoting social and technical progress. Making the results of scientific research readily available to other research groups (and the public at large) stimulates fact-based discussion, facilitates verification of reproducibility, and empowers others to build upon previous results. Public dissemination also fulfills an educational role, by enabling interested parties to become familiar with research without requiring direct participation in scientific process. The assumption that scientific literature is only intended for the scientific community is one that does not acknowledge the responsibility that scientific research has to society at large, particularly in the cases where research has been made possible using public funding. For example, patient advocacy groups are requesting greater access to the results of medical research, they argue persuasively that when research is paid for with public funds the results need to be available to the public.

1.4.1 Open Access Journals

When many people think of Open Science and initiatives to promote it, Open Access Journals are typically first on the list. Open Access Journals, are defined as scholarly journals available online ”without financial, legal, or technical barriers other than those inseparable from gaining access to the internet itself” [13]. For many, Open Access simply means publishing the results of scientific research in journal form, paying for publication either by charging the author(s) a fee to publish, or asking the authors to absorb the cost by self-archiving (or publishing) journal articles on their own web site. Open Access journals are becoming quite popular, and there has been a flurry of new journals in the last few years [5].

While an important first step, this simple view of Open Access as an open journal does a disservice to the cause of Open Science. Publishing a journal article, no matter how easy the access nor small the cost, does not guarantee reproducibility. Without associated data (Open Data) and methods (Open Source), experiments described in an article typically cannot be easily reproduced. Thus many Open Access journals also require submission of data and source code (see the 1.6.3 later in this chapter for more details).

There are other interesting features that Open Access journals provide including version control and review, as described in the following subsections. Another important aspect that is often neglected is the choice of licensing, where some open access journals prevent commercial use, or derivative work thereby blocking important reuse of published articles.

1.4.2 Versioning Documents

Similar to the arguments made in the previous section on software testing (Section 1.3.2), errors are pervasive and to be expected throughout complex endeavors such as scientific research. Accepting this reality, and controlling it, requires a continuous process aimed at identifying and correcting errors. Consequently, the venues used for sharing scientific information and disseminating results must provide mechanisms for capturing community feedback, tracking changes, and providing access to current documents as well as the previous versions.

1.4.3 Open Reviews

The open-minded experimentation around Open Access journals offers an opportunity to reconsider many of the longstanding practices of scientific publishing, some of which have become long time traditions and deserve to be revisited given the emergence of the Web.

One of the key aspects in which open access journals can improve communications in the scientific community, is the modification of the typical publishing workflow. In traditional journals, publication is delayed until after articles have been vetted by reviewers. This painstaking process can take years from the time of first submission to the time of publication. This delay greatly diminishes the value of the final publication, particularly in topics that are related to the rapidly evolving domain of computational research. The traditional review process also privatizes the conversation between the authors and reviewers, and by doing so deprives the community of valuable discussions and from the benefit of observing scientific discourse.

An alternative to the traditional closed-doors, anonymous peer-review process, is the practice of open reviews. This is a practice inspired by the self-regulation and self-certification processes that many online communities have adopted to curate their materials and to perform quality control on their content [21].

Open reviews blur the distinction between readers and reviewers, since they both have access to exactly the same amount of material. A reviewer is simply a reader who feels compelled and motivated to provide feedback to the authors. This is in contrast to traditional reviews that are performed by a select group of individuals who are considered to be experts in a domain. The notion of a *peer* in an open source community is anyone who participates; distinctions are made based on contributions; and authority is defined by meritocratic recognition.

Open reviews more directly honor the concept of *Peer Review* by empowering all our peers, not only a narrow group of selected experts, to share their views on the content of published materials. By not relying on the authority of experts, open reviews are better aligned with the tenet of the scientific method: "*to withstand the domination of authority and to verify all statements by an appeal to facts determined by experiment.*" [7].

1.5 Open Standards

Open Science is usually described as requiring three basic components: open data, open access, and open source. Providing these elements is enough to reproduce an experiment assuming that all information is provided. However many practitioners of Open Science also advocate for a another element: Open Standards. If reproducibility is the goal, why is this additional element important?

There are several answers to this question. Pragmatically, using standards, or helping to create them, is an indication that a researcher is earnest about sharing and hence practicing Open Science. It may be true that open data, access and source enable reproducibility, they do not necessarily make it easy. Using standards generally results in more efficient science as information can be readily accessed and analyzed, making life easier for other researchers. Open Standards also enable large-scale analysis in which multiple contributions are combined to form new insights, or build new tools. Consider the following examples:

standards simplify access to multiple data sets from different research groups which can be combined to support analysis of larger information pools. Open document repositories can be analyzed (using methods from text analysis) to identify emerging concepts and determine relationships between lines of research; such information is important to science as well as investors and technology managers. And finally, well designed and implemented code can be reused and combined to build powerful and useful software.

There is another way in which Open Standards support the scientific mission. That is, to ensure reproducibility it is important to run experiments under controlled conditions. Therefore Open Standards can also be thought of as data, software systems and/or publications which are certified at a known state. This enables researchers to build on well-defined foundations, thus a particular Open Standard specifies one of the components composing the environment of an experimental process; for example standard data sets, software libraries and even computing platforms. Such control is necessary when comparing algorithms or otherwise evaluating the performance of a computational system. For that matter, even supporting laboratory software used to acquire data may produce different results under the same conditions if not carefully controlled. Therefore, using Open Standards can remove experimental uncertainty.

It is not possible to say exactly what standards to use. Different research fields, ontologies, data composition, and software systems require different standards to support research and foster sharing. Moreover as knowledge expands, standards must evolve as well. Therefore the use of standards is a delicate balance between the demands of innovation and the requirements of sharing. However, it is important to distinguish between standards that are open and those that are not.

Open Standards promote sharing and support the scientific mission. Many non-open standards (which may claim to be open) permit reasonable and non-discriminatory patent licensing fees which erect barriers to sharing and hence reproducibility. Generally Open Standards are developed by collaborative teams, use permissive licensing free of licensing entanglements, are thoroughly documented with reference implementations, and are meant to be widely used [14]. An interesting twist to some forms of Open Standards licensing is that predatory embrace-and-extend tactics may be prohibited to prevent organizations with influential control over a market or technology area to game implementations and impose restrictions on how others use the standards [33].

1.6 Open Science Platform

In the previous sections we described many of the motivations and basic concepts that drive the practice of Open Science. In this section we provide concrete details of several key components that constitute our daily practice and workflow.

1.6.1 Midas Platform

As discussed earlier in this chapter, data-centric computing is critical to the practice of Open Science (Section 1.2.2). For many of our applications we use the Midas Platform [28, 27], which is an integrated, open-source toolkit that enables the rapid creation of customized, integrated applications with web-enabled data storage and management, advanced visualization, and processing (see Figure 1.3). The Midas Platform is implemented as a modular PHP framework with a variety of backend databases (in particular PostgreSQL, MySQL and non-relational), that scales well to large data.

The Midas Platform system can be installed and deployed without any customization, it has been designed with this capability in mind. Given that data-centric computing depends on diverse workflows and it is generally custom-integrated depending on the needs of a project, there is no single solution that fits all possible applications. Therefore the Midas platform supports additional extension mechanisms such as plug-ins and layouts to facilitate customization.

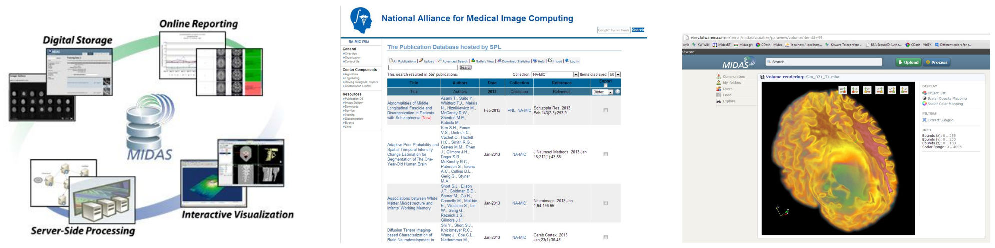

Online Reporting
Digital Storage
MIDAS
Server-Side Processing
Interactive Visualization
National Alliance for Medical Image Computing
The Publication Database hosted by SPL
MIDAS
Volume rendering

FIGURE 1.3: Midas is an open source platform supporting data-centric computing (left). It has been used in a variety of data-intensive applications, including publication databases (which include data and images—center) to advanced volume rendering (right).

Some example customization efforts have led to the implementation of several different types of document database (see Figure 1.3) including the Optical Society of America's Interactive Science Publishing system [15] and the Insight Journal (described later in Section 1.6.3). The Publication Database is a specialization of the platform to support academic publications, for example at the Surgical Planning Lab at Harvard Brigham & Women's Hospital, the NA-MIC project [19] uses the Publication Database to host content (papers, data and images) from all contributors to the project [11]. The Publication Database is a digital repository for scientific papers and a computational infrastructure intended to facilitate the outreach activities of scientists. It provides a streamlined way to upload, present and share the research and publishing activity from an institution. This is an example of a resource that can be used to implement *Institutional Repositories* and provide the mechanisms for practicing Open Access.

1.6.2 CMake-Based Software Process

As described earlier, the effective practice of Open Source depends on a rigorous software process. Our process relies heavily on the CMake, CPack, CTest, and CDash family of tools [30], which we have organically developed and refined over many years of developing large-scale open source projects. In addition, we prefer the git DVCS; although we continue to use SVN (and other VCS such as CVS when necessary). Basically the software process we use is highly automated, closed-loop, and convergent (Figure 1.4)—this is vital to ensuring the stability of the software. As shown in the figure, the software repository is constantly monitored for additions, and when changes occur the software is tested and the results displayed on a software-quality dashboard (Figure 1.5). Developers and users monitor the dashboard and correct any errors as necessary, pushing code changes to the repository and completing the cycle. The process runs continuously and hence ensures reproducibility and informs users of the system of problems in a timely manner.

In the following subsection, we describe the software process in more detail. Along the way we refer to several systems including the Visualization Toolkit (VTK) [20] and the Insight Segmentation and Registration Toolkit (ITK) [9]. These are examples of large-scale software systems that rely on formal code review, with active communities of thousands of members, and decades of use.

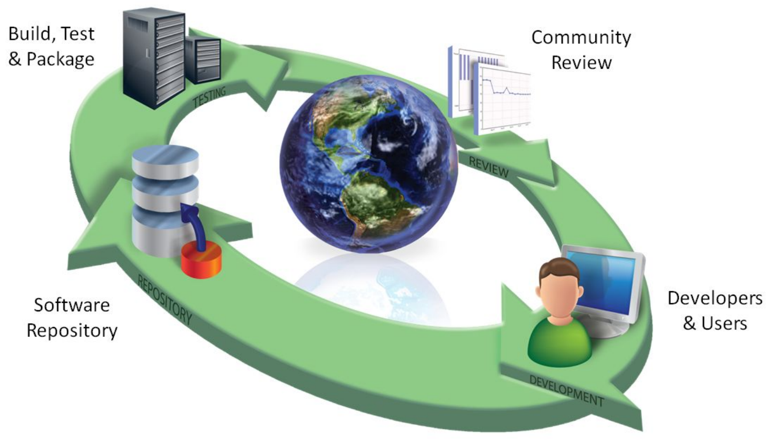


FIGURE 1.4: A closed-loop software process depends on the CMake family of tools. CMake is used to build software across multiple platforms. CTest tests software on a particular client platform; CDash receives such test results and displays them on a Web-based dashboard visible to the community. Finally, CPack is used to package and distribute code and executables for multiple platforms.

1.6.2.1 Overview

CMake is an open source tool for building complex software systems across multiple computing platforms. As the platform consists of various combinations of operating system, hardware, and system libraries, CMake manages this complexity in a relatively transparent way. Using CMake requires specifying dependencies on third party packages, and selecting options to enable or disable certain features and behaviors of the software package in question. By embedding this information in CMake scripts, it is possible to standardize the process of configuration for many different platforms and to store such rules along with the source code.

CMake itself does not perform builds, but instead focuses on the configuration process that will produce standardized builds. In particular, CMake generates native project build files according to the platform, for example: Unix Makefiles, XCode, Visual Studio, Ninja or Eclipse. In this way the rules written in build system files are carried along with the project, and are maintained and tracked in the same version control system that the project uses.

The use of CMake facilitates the sharing of software for scientific research by empowering developers to configure software to run on a variety of platforms that range from embedded systems and laptops through to supercomputers. Examples of packages that use CMake include KDE, LAPACK, CLAPACK, ParaView, Trilinos, VTK, and ITK, which are a few of the thousands of software projects using CMake [2].

CTest is a companion tool to CMake, it is also open source and is distributed as part of the CMake package. The goal of CTest is to facilitate the process of running tests and reporting their outcomes to centralized sites. The daily use of CTest is quite simple. It is reduced to scripting the command line instructions that one would have used to run the test manually. However, in the process of scripting it, the developer must face the questions of:

- Where is the input data?
- Where to generate the output data?

- What parameters are necessary to execute programs?

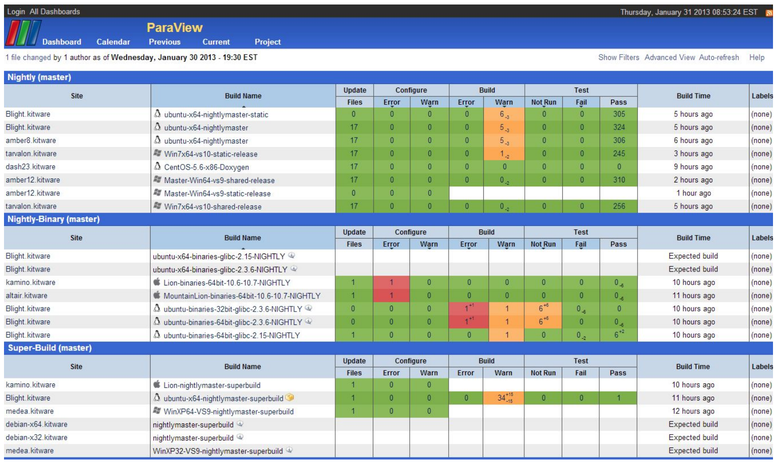
The fact that open source developers confront these questions on a daily basis forces them to be quite organized and methodical. They must figure out how to refer to data regardless of the particular computer system that is being used to run the test.

A key entry in the driving CMakeLists.txt file are the commands that describe tests that will be run later with CTest. A typical entry looks like:

```
add_test( executable input1 input2 output parameter1 parameter2)
```

This includes the location and identification of the input data, and the fully defined set of parameters required to run the test. It turns out that this also provides documentation for the test itself, at a level of granularity that is rarely found in scientific publications.

CDash is a web-tool that collects and summarizes the results of the CTest testing process across multiple platforms. The project dashboard (Figure 1.5) provides a rich set of hyperlinks that supports rapid navigation through the output of the build process, and even into the source code if necessary. Hence compile errors, or test failures, are easy to find and analyze. There are also many filtering options that make it possible to, for example, determine exactly when a test started failing, which in combination with the information provided by the revision control system to track changes in the code, is invaluable when determining what change caused a failure displayed on on the dashboard.



1 file changed by 1 author as of Wednesday, January 30 2013 - 19:30 EST

Nightly (master)

Site	Build Name	Update	Configure		Build		Test			Build Time	Labels
		Files	Error	Warn	Error	Warn	Not Run	Fail	Pass		
Blight.kitware	ubuntu-x64-nightlymaster-static	0	0	0	0	6_{-3}	0	0	305	5 hours ago	(none)
Blight.kitware	ubuntu-x64-nightlymaster	17	0	0	0	5_{-3}	0	0	324	5 hours ago	(none)
amber8.kitware	ubuntu-x64-nightlymaster	17	0	0	0	5_{-3}	0	0	306	6 hours ago	(none)
tarvalon.kitware	Win7x64-vs10-static-release	17	0	0	0	1_{-2}	0	0	245	3 hours ago	(none)
dash23.kitware	CentOS-5.6-x86-Doxygen	17	0	0	0	0	0	0	0	9 hours ago	(none)
amber12.kitware	Master-Win64-vs9-shared-release	17	0	0	0	0_{-2}	0	0	310	2 hours ago	(none)
amber12.kitware	Master-Win64-vs9-static-release	0	0	0						1 hour ago	(none)
tarvalon.kitware	Win7x64-vs10-shared-release	17	0	0	0	0_{-2}	0	0	256	5 hours ago	(none)

Nightly-Binary (master)

Site	Build Name	Update	Configure		Build		Test			Build Time	Labels
		Files	Error	Warn	Error	Warn	Not Run	Fail	Pass		
Blight.kitware	ubuntu-x64-binaries-glibc-2.15-NIGHTLY									Expected build	(none)
Blight.kitware	ubuntu-x64-binaries-glibc-2.3.6-NIGHTLY									Expected build	(none)
kamino.kitware	Lion-binaries-64bit-10.6-10.7-NIGHTLY	1	1	0	0	0	0	0	0_{-6}	10 hours ago	(none)
altair.kitware	MountainLion-binaries-64bit-10.6-10.7-NIGHTLY	1	1	0	0	0	0	0	0_{-6}	11 hours ago	(none)
Blight.kitware	ubuntu-binaries-32bit-glibc-2.3.6-NIGHTLY	0	0	0	1^{+1}	1	6^{+6}	0_{-6}	0	10 hours ago	(none)
Blight.kitware	ubuntu-binaries-64bit-glibc-2.3.6-NIGHTLY	0	0	0	1^{+1}	1	6^{+6}	0	0_{-6}	10 hours ago	(none)
Blight.kitware	ubuntu-binaries-64bit-glibc-2.15-NIGHTLY	1	0	0	0	1	0	0_{-2}	6^{+2}	10 hours ago	(none)

Super-Build (master)

Site	Build Name	Update	Configure		Build		Test			Build Time	Labels
		Files	Error	Warn	Error	Warn	Not Run	Fail	Pass		
kamino.kitware	Lion-nightlymaster-superbuild	1	0	0						10 hours ago	(none)
Blight.kitware	ubuntu-x64-nightlymaster-superbuild	1	0	0	0	34^{+15}_{-15}	0	0	1	11 hours ago	(none)
medea.kitware	WinXP64-VS9-nightlymaster-superbuild	1	0	0						12 hours ago	(none)
debian-x64.kitware	nightlymaster-superbuild									Expected build	(none)
debian-x32.kitware	nightlymaster-superbuild									Expected build	(none)
medea.kitware	WinXP32-VS9-nightlymaster-superbuild									Expected build	(none)

FIGURE 1.5: A portion of a CDash dashboard (from the ParaView open-source project). The dashboard color codes errors and warnings, and is heavily hyperlinked to provide access into the input and output of the build process.

Finally, CPack is used to automatically package and distribute software releases across multiple operating systems. This greatly simplifies the release process and enables frequent, rapid releases of software. This supports the Open Source tenet of "*Release Early, Release Often*" by which software is often released on a daily basis.

1.6.2.2 Unit Testing

Unit testing is a software engineering practice that focuses on creating tests for the smallest possible functional units of the software being developed. This makes it possible to locate errors with a high granularity when they are introduced into the software.

In the particular case of ITK and VTK, which are object-oriented C++ libraries, the practice of unit testing is tightly coupled with the design and implementation of classes and their methods. In ITK we start by writing an empty C++ class with something similar to the following pseudo-code:

```
class itkNewImageFilter
  {
  public:
  };
```

and a test for it in the simple form

```
int main(int argc, char *argv[])
  {
  itkNewImageFilter filter;

  return 0;
  }
```

Then we would add a piece to the test:

```
int main(int argc, char *argv[])
  {
  itkImage inputImage;
  itkNewImageFilter filter;
  filter->SetInput( inputImage );
  return 0;
  }
```

and then proceed to implement such method in the class:

```
class itkNewImageFilter
  {
  public:
    void SetInput(itkImage image) { this->SetInternalImage = image; }
};
```

This may appear to be an agonizingly slow way to write software, but in practice it is the fastest way to write software that *does not have to be rewritten.* It is a common mistake for developers to go in long stretches of writing hundreds or even thousands of lines of code, and then as an afterthought, attempt to write tests for them. The consequence is that by the time they start writing tests, they have already introduced many bugs and inconsistencies in their code. Such defects now have to be found and fixed through the much more expensive and laborious process of detective work. The average density of errors in the software industry is one bug for every thousand lines of code[1] [4].

This is with the caveat that during the debugging process new bugs will possibly be introduced. It is known that about fifty percent of all bugs are introduced while the developer is trying to fix other bugs [26]. These *second generation* bugs are the beginning of a nearly endless task, because again, attempts to fix either one of those bugs will, half of the time, introduce *third generation* bugs, and so on. The mathematically inclined readers would already be estimating that one original bug becomes $\sum 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots$ bugs.

The methodical process described is at the same level of rigor that one would expect

[1]The average bug density of Open Source projects is 0.45 defects per thousand lines of code [4]

from any well-trained experimental researcher. Therefore, there should not be any objections to the cultural adoption of these practices when developing software for research applications. To put it bluntly: if a research software developer does not have the discipline to write unit tests, then they are also likely to lack the discipline to be a well-qualified experimental researcher. Once again, this is not a technical challenge but rather a cultural challenge. To incentivize reproducibility in scientific research, it is therefore necessary to work simultaneously on multiple fronts. In particular, providing technical tools, while at the same time ensuring specific behaviors are celebrated or condemned through the culture of a community.

1.6.2.3 Examples of Code Review

The ITK and VTK projects use Gerrit (which depends on the DVCS git) along with a simple evaluation script to quickly check a proposed change for basic correctness. Meaning, the script enforces certain guidelines such as style and naming conventions including inappropriate white space, appropriate line length and termination delimiters, and hard-coded path names. In addition to these checks, the events generated by Gerrit are monitored by another system which submits a build request to an automated build farm if a developer is in the core group of developers. This initial build request may be a subset of the entire test suite in order to enable a quick turn around. This build test utilizes a system called CDash@Home to request a build of the proposed change on Linux, Windows and Mac OS X [17, 3] systems. Hence the automated check-in evaluation process not only verifies the project successfully builds on these common computing platforms, but also runs some quick tests and submits the results (Figure 1.6).

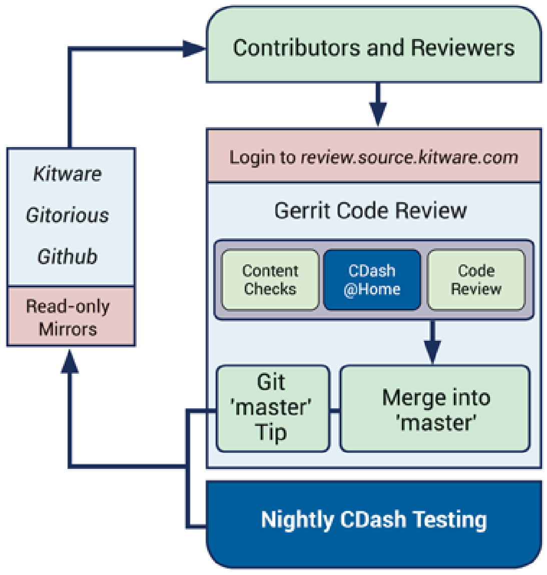


FIGURE 1.6: Graphical overview of the software process that incorporates Gerrit for code review, CDash@Home for pre-testing and CDash for nightly testing. Note the difference between those with write access and those without is reduced.

As a result of this initial smoke test, reviewers can view the build on the set of core supported platforms and compilers to check for any serious regressions, freeing them to concentrate on reviewing the substance of the change. Once merged into the main development branch a larger number of machines download the new version of the code and proceed to perform more comprehensive tests. This practice of reviewing and testing of patches before they are even merged into the main code base enables us to maintain much higher stability on the main development branch than was previously possible, and also better engages the community in the maintenance effort. The result is to significantly blur the once sharp line between committer (a developer with commit rights) and budding contributor (someone who is just beginning to learn and contribute to a project).

1.6.3 The Insight Journal

The Insight Journal is an open technical journal built on the principles of Open Access, Open Data, and Open Source [18]. This on-line journal focuses on the domain of Medical Image Computing, and enforces the verification of reproducibility for all contributed articles. The Insight Journal went online in 2005, thanks to generous funding from the National Library of Medicine at the National Institutes of Health. The Journal began as an effort to facilitate the sharing of image analysis algorithms in support of the ITK community. Today, there are several derivatives of the Insight Journal, such as the VTK Journal, the Midas Journal, and the OSEHRA Technical Journal, in use by other communities.

The creation of the Insight Journal was a response of the ITK developer community that recognized a large number of papers published in the medical imaging field were not reproducible. While this is unfortunately common across other disciplines too, it was particularly frustrating to the development team of the ITK open source software library. Initially, the team naively believed that published papers would have an associated open implementation necessary to produce useful results. Unfortunately, the culture of openness and verification in the open source world collided with the failure of reproducibility that too often occurs in scientific research. The ITK development team found that for many algorithms, their publication in journals were too often just general guidelines to the overall flow of the algorithm, and that the authors failed to provide a reproducible implementation covering all facets of the technique necessary to implement a working implementation.

From this experience, the community decided to create a journal of the type that would have been useful to the initial development of ITK. Such an ideal journal would require article submissions that included functioning source code, as well as tests and examples demonstrating the use of the code. These tests and examples further required the inclusion of all input data; and to support comparisons, the inclusion of the output data generated by running the contributed code. Finally, for each run of a test or an example, the article would include a full specification of the parameters necessary for it to run. The whole submission package, including the article, source code, tests and data, would be available in its entirely to reviewers and readers of the journal under permissive licenses enabling them to download, use, modify and redistribute the materials from the journal without having to involve the legal departments of their respective institutions. Based on these requirements, the Insight Journal was created.

The Insight Journal, and other similar open access journals, fill a gap in the practice of scientific research by providing a venue where practitioners can share working versions of research code in a usable way. Despite the fact that the Journal does not fit the traditional academic publishing model, that is mostly oriented to support career evaluations, it has become a key element of the ITK ecosystem. Running continuously for seven years, it has (at the time of this writing):

- 3,904 registered subscribers
- 540 published articles

- 821 reviews

The usefulness of the Journal, as a vehicle for sharing contributions with peers, has been found to be extraordinary, although it currently does little to help academics score points essential for progression and tenure. However, since it enforces the verification of reproducibility, it is a real scientific journal that successfully facilitates communication across the research community, accelerating scientific progress by minimizing publication delays, and providing an environment necessary for subscribers to use it in their own research. It is quite common for rapid dialogues to emerge between researchers, and for members of the community to express appreciation at finding leading-edge computational tools, with associated data and documentation, which address their current challenges.

One of the major features of the Journal is that it takes advantage of the near-zero costs to store and transmit data in today's networked world. In particular, it has eliminated most of the publishing restrictions that many traditional journals have inherited from the age of the printing press, including: page limits, restrictions on number and type of figures, problems updating revisions, use of color, limitations on supplemental materials, and long turn-around cycles.

1.6.3.1 Practical Details

The Journal follows well established practices of open source communities which are rooted in continuous openness and transparency, and in particular heeds the mantra *Release Early, Release Often*. As a result, papers are published within 24 hours of submission, allowing time only to remove spam submissions, followed by an open review process that is publicly visible to the entire community. This public process elevates the civility of the review dialogue while greatly accelerating access to the material contributed by the authors.

When we began designing the Insight Journal in 2005, one of the first concerns we had can be described this way: *We are inviting people on the Internet to send us arbitrary source code that we are going to compile and run on our machines*. It did not take long before we realized that an encapsulated environment was required to run these source code contributions in a secure way. The solution was implemented using the Xen virtualization platform [22], along with a process to launch a virtual machine on demand whenever an article was received by the Journal. Thus a Web-based front-end triggers a request to launch a pre-configured virtual machine with the installed software tools and platforms required to run the code accompanying the submission. For example, the pre-configured VM has several recent versions of ITK, VTK, and CMake installed. A mechanism is provided to authors to specify the versions of ITK, VTK and CMake required to build their submitted code. Automated scripts then take the source code from the submitted article package, copy it into the virtual machine, expand it, configure it, build it and run the tests submitted by the authors. The results of the submitted tests are then posted as an initial, automatic review to the Journal. In this way, readers are primed with the initial information as to whether the Journal infrastructure was able to build and replicate the results that the original authors described in the submission.

Given that authors can also submit revisions to their articles, along with modifications to the code and data (without having to go through editorial hurdles), the entire process unfolds in a rapid and agile manner. As soon as an article is submitted, notification is sent to all subscribers (about 3,900 people), and to the ITK community mailing list (over 2,300 people). The article is then made available for download, including the PDF document, all source code, test code, examples, and input data required to verify the content of the submission, as well as the output data resulting from executing the software on the input data. The goal is to facilitate reproducibility in a very pragmatic way, empowering any reader of the Insight Journal with the ability to rerun the experiments described in the article, with minimal effort, and verify or build upon the research described.

Any subscriber to the Journal is able to contribute reviews to the article. The reviews are non-anonymous and are posted publicly. This spurs an open conversation in which the reviewer and the author(s) exchange views and ideas, point out errors, and suggest areas that could be improved. The entire community benefits from being exposed to the conversation captured within the Journal web site. By encouraging all members of the scientific community to participate, we cultivate an engaged and participatory community where we all share the responsibility and the opportunities for moving the science forward.

1.6.3.2 Community Involvement

From its inception, the Insight Journal encouraged authors to submit revisions of their papers, with corrections and ongoing improvements. Being free of the limitations of publishing on physical paper, we had the ability to correct any errors by simply allowing and encouraging authors to submit subsequent modified versions of their PDF documents and/or their source code, data and configuration. This created a working environment suitable for spurring collaboration across the community.

The process was quite successful, and as the Journal became more popular, readers and authors started to have conversations that led to improvements in the source code contributions. As this happened, it rapidly became evident that the process of uploading modified versions of the articles and source code, even though it was far more flexible than the traditional paper-based publishing venues, was too cumbersome when compared to other well-known agile open source processes. More specifically, open source projects routinely make modifications to their source code using version control systems (see Section 1.3.1. To honor this tradition, a second generation of the Insight Journal was put in place[2], where every code contribution submitted to the Journal was automatically inserted into a back-end Git repository[3].

The Insight Journal has dramatically collapsed the time from submission to publication, which has been enthusiastically embraced by the community. What used to be an arduous publication cycle of two to five years, now takes minutes with full disclosure. Not only is the code available almost immediately, but it is stored in an infrastructure that permits further development, improvement and maintenance of the data, publication, and code.

1.6.3.3 Data Concerns

As the Insight Journal was adopted by the ITK community, it became clear that sharing data was more challenging than first thought, and required subsequent modifications to the sharing process. Initially we made it clear that licensing restrictions were to be minimal. We prefer that data is licensed using CC0 and similar non-reciprocal licensing models. One of our goals is to open up whole new fields of data reuse and meta-analyses. For example, we envision large analyses traversing hundreds or thousands of papers (and their data) to spot wider trends that the original researchers may have overlooked. This requires published work that uses open access licenses and enables data mining with semantic meaning encoded, and provides open APIs using open standards such as REST and XML/JSON to encode the results.

In addition there were challenges managing data. Some of the issues we addressed include:

- large data set size,

[2]`http://www.kitware.com/blog/home/post/167`
[3]`https://github.com/midas-journal`

- collections with a large number of data sets, and
- limitations on access.

To address those issues, the ITK community created a data access solution based on the Midas Platform (see Section 1.6.1). Some of the more important features include:

- Data revision control, based on content,
- An API for downloading data on demand,
- A mechanism for uniquely identifying data sets to be downloaded, and
- A mechanism for sharing data stored on local disks (for performance).

This collection of features enabled complex computational research scenarios such as the following.

- A research group gathers a data collection and uploads it to a database.
- The upload process generates unique identifiers for every data set, based on its content.
- A second research group decides to use this collection as input for a data analysis task.
- CMake scripts are written, which refer to the specific data sets to be used as input, and assign them to the specific executables of a computational experiment.
- An experiment is run by this second research group, which automatically downloads the data shared by the first research group and uses it as input to its computation.
- Finally, a third research group takes the source code and configuration provided by the second research group, and replicates their experiment by building executables from the source code and downloading the original data shared by the first group.

With this infrastructure, it is possible to take a set of algorithms and run them rapidly on multiple data collections, a task that could have conceivably taken years of effort in the absence of such a computational platform employing the principles of open source, open access, open data, and open standards.

1.6.4 Scalable Computing

Modern science relies heavily on computation. Analytical processes can be used to tease relationships from data. Often theories are simulated on a computer and compared with experimental results. Even the process of acquiring measurements relies heavily on computers: consider the image and signal processing that goes into observing stellar phenomena.

In our practice of Open Science, we do not rely on any single approach to perform computation. Typically we employ open source systems like Midas, VTK, and ITK to build custom applications. However, there are certain systems that we use when data becomes large and complex, or we need extra computing resources. We describe these systems in the following section.

1.6.4.1 High Performance Computing

Throughout the world researchers are increasingly turning to high-performance computing (HPC) to conduct their work. More often than not this means making use of dedicated HPC resources which often have unusual computing environments. One of the major challenges is developing cross-platform software that can run on these often specialized platforms. For example, some supercomputers do not provide graphics hardware, which means that applications that depend on OpenGL have to use software rendering. Further, the individual nodes of an HPC resource often run a limited version of the operating system (typically Linux), and as a result significant work is necessary to port code designed for off-the-shelf desktop operating systems to work on such resources. These and other challenges are only going to become more pronounced as HPC moves towards exascale computing [31], in an environment where computational FLOPS are cheap (e.g., millions of computing cores) and I/O and data transfer are expensive (in terms of performance and energy costs).

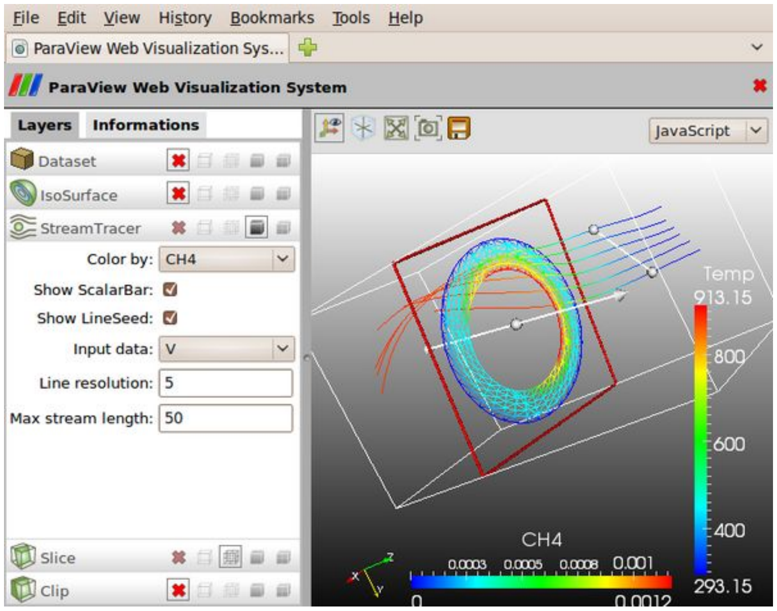
The inherent complexity and challenges of HPC means that open source software is essential to advancing the state-of-the-art. There are several reasons for this:

- Open source software can be more easily adapted to HPC platforms. There are minimal licensing issues and software engineers have full access to the code, which they can modify to fit to the platform.
- Problems can be more easily discovered and corrected since code is not hidden. Debugging tends to be much easier. This is particularly important as advanced parallel-computing algorithms use complex distributed and shared-memory techniques to maximize the performance of HPC resources.
- Computing time on these machines is typically limited and expensive, therefore carefully controlled, which makes verification of the correct operation of the code more important than ever.
- Commercial code is often licensed on a per CPU-core basis (or similar). With an explosion in the number of computing cores, the pricing model of commercial software causes dramatic increases in cost. In contrast, open source software does not carry this burden.

In the following we describe some of the HPC software that we use in our practice of Open Science. All of the open source systems listed below use permissive, non-reciprocal BSD licenses.

VTK is a C++ toolkit (wrapped in Python, Java, and Tcl languages) developed by a large community of international contributors. It originated as companion software to a book on 3D visualization [20]. Now nearly 20 years old (development started in 1993) with millions of lines of code, it has served as a foundational computing tool for 3D graphics, scientific and data visualization, computational geometry, human-computer interaction, informatics, image and volume analysis, engineering simulation, and more. The system is inherently portable, and has been run on systems ranging from the Raspberry Pi to some of the largest supercomputers (at the scale of hundreds of thousands of processors).

ParaView is an open source, large-scale parallel visualization application leveraging VTK to provide visual data analysis for many data sources, including computational fluid dynamics, medical computing, engineering simulation, combustion, point clouds (from LIDAR or other imaging sources), climate simulation, video processing, and so on [23]. ParaView employs an advanced client-server computing architecture that enables light weight clients to connect with computing and/or graphics servers residing on an HPC platform. Typically run using distributed, parallel computing model, it can also leverage large shared-memory parallel systems.



Color by: CH4
Show ScalarBar: ✓
Show LineSeed: ✓
Input data: V
Line resolution: 5
Max stream length: 50

FIGURE 1.7: ParaViewWeb enables advanced, large data visualization capabilities through web clients, which in turn coordinate with a ParaView server which may reside on an HPC system. It also supports collaboration across multiple, simultaneous viewers.

ParaViewWeb is a client application providing a collaborative, remote web interface for 3D visualization using the ParaView server [16]. It also provides a JavaScript API for ParaView scripting, features and capabilities. ParaViewWeb has been designed so that advanced visualization tools can be easily integrated into a web page, and multiple viewers can simultaneously view, interact with, and collaborate around data (Figure 1.7).

Catalyst is a data analysis and visualization library designed to be tightly coupled with simulation codes [1]. It was created in response to the unfortunate reality that HPC systems produce too much data to be fully captured (due to IO and disk limitations), thus co-processing systems like Catalyst are embedded into the computing process to analyze and extract only essential data during computation. This also makes it possible to monitor long running analyses, and control them during execution.

MoleQueue is an auxiliary application used to launch, monitor and control HPC resources. Managing HPC systems is still a challenging task and MoleQueue makes it much easier by abstracting many of the differences between remote resources [10], and providing a simple API for client applications on the local desktop machine.

1.6.4.2 Science as a Service

In recent years cloud computing has become an increasingly important part of scientific computing. Compared with HPC, cloud computing systems are quite similar in that the resources are time limited, and they often run a lean operating system. Thus to create appropriate computational resources distributed memory approaches must generally be employed. Indeed one popular open source package for scientific computing on the Amazon EC2 offering is Star Cluster, which simplifies the process of configuring and deploying a Sun Grid Engine cluster on the Amazon platform that closely resembles a typical batch-scheduled HPC platform.

Once deployed, similar approaches to HPC can be used to schedule jobs, and communicate between nodes. The communication between nodes is typically slower than purpose built supercomputers, but time can be more easily purchased, and in some cases the elasticity of the resource can be an enormous asset. Going forward it is clear that the cloud will be a major part of the market for reproducible science, offering some unique opportunities.

Cloud providers, such as Amazon and Rackspace, make it relatively easy to customize the operating system running on one cloud instance and then deploy clones of it on one or more instances. These images can be shared publicly with others, enabling developers to produce pristine reference images of a full operating system where correct operation of the code has been tested. This means that others are able to verify results by purchasing time on the platform, deploying an instance with the reference image and duplicating the reported results. When coupled with open source codes and the Linux operating system there is no restriction on distribution and even non-experts have the opportunity to use complex codes where compilation, configuration and deployment can take significant amounts of time, and have now been made available to them ready to use.

The current widespread availability of cloud computing resources provides the opportunity to implement a reproducibility verification computational platform in a highly scalable way, without having to own and maintain the resources. By taking advantage of the network effects and the economies of scale, a full-fledged scientific computational platform is available at a cost that is very close the marginal cost of using the raw computational resources. A new scale of scientific research is made possible by these platforms, that will empower the computational research community to ask ambitious questions without having to add to their budget the full cost of large-scale resources. Large computational experiments are no longer the exclusive privilege of institutions that can afford the acquisition, installation, and maintenance of large computational resources such as clusters and supercomputers. Instead, it is now available to anyone, for the cost of the resources that are actually used during a given experiment.

As a result, another interesting development enabled by Open Science on the cloud is the development of a commercial marketplace for science [24]. The computational platform can be offered as a service to verify the reproducibility of reported results provided by a neutral third party. Such an approach is a complement to the [6], where a market-based system has been put in place to enable interested parties to contract services to replicate experiments from a set of trusted service providers. By delegating the experimental verification to organizations that have the suitable infrastructure, and that have a good reputation of being neutral and objective, opens up new possibilities in the practice of verification of experimental results, at a lower cost, thanks to a better allocation of resources and an open marketplace.

A typical scenario is for a pharmaceutical company to contract a *reproducibility verification service provider* to reproduce a set of bioinformatics experiments that they may have come across in a scientific publication. The original authors would have made available, as part of their publication, all the materials required to replicate their computational experiments, including source code, data and configuration parameters. The *verification provider*, who has a pre-configured and scalable computational platform, can then proceed to rerun those experiments and report back to its customer on the outcomes of these experiments. In terms of computational costs, the customer would only have to cover the cost of cloud resource usage incurred during the execution of the experiments, with no need to own and maintain the full computational platform.

At Kitware we are currently experimenting with these and a variety of other Open Science practices in the cloud. We envision providing our advanced, open source software tools to host data, support data-centric computing, and facilitate the sharing of scientific knowledge.

1.7 Challenges

The advent of the scientific method in the 17th century has enriched society in profound ways, from improving health to providing a multitude of goods and services to offering fundamental insights into the workings of the physical universe. Open Science ensures that this legacy of innovation and understanding continues to address the challenges facing us in the 21st century and beyond. However, supporting Open Science comes at a significant cost, ensuring reproducibility requires resources for sharing and to nurture communities. Additionally, guaranteeing reproducibility in an ever-changing computing environment is difficult. And finally, with human reputation, recognition and achievement on the line, we need to rethink the ways that scientists are evaluated and rewarded. These topics are addressed in the following paragraphs.

As described earlier, the size of data produced in science is growing at an enormous rate. Billions of dollars are spent to acquire or compute it, hence it represents a scarce resource which cannot be easily replaced. Once it is collected, large data is expensive to store, provide access to, move, and analyze. In the past, data was often tabulated in paper publications and stored in a library, now sophisticated data centers (including in some cases high-performance computing support to process it) are required. This poses a problem in that very few institutions have the computing resources, or the wherewithal, to support such large-scale, data-intensive science. Fortunately computing solutions are emerging (such as Amazon's EC2, EBS, and Glacier) but it remains to be seen whether commercial vendors are committed in the long-term to supporting scientific data. Supporting aging data is problematic as the justification for maintaining it wanes with the perception of declining value.

Another insidious problem is the shifting sands of the computing environment. Whether it is old software written in a programming language that has evolved or become obsolete, or the computing platform (on which the software executes) which includes the operating system, software libraries and hardware; computing environments change rapidly and play a major role in the reproducibility challenge. It is conceivable that software and data written in a certain era may no longer execute on future platforms. While open standards and commitment to backward compatibility do much to address this problem, computing environments have become so complex that it is hard to imagine indefinitely maintaining a reproducible configuration. Proposed solutions go so far as to propose virtual machines which are stored along with scientific software; however there is no guarantee that future platforms will support existing VMs, and future computing architectures may be drastically different including high degrees of parallelism and based on distributed Web resources.

Despite these technical obstacles, the biggest challenge may be addressing the entrenched scientific institution, social norms and the way its various members and organizations interoperate. As described previously, the scientific publishing community is under siege due to their out-of-touch business model in the era of the Internet. Yet bigger issues remain including the tendency of some scientist's to be overly protective of their work (mostly due to the way they are evaluated for career rewards), which interferes with collaboration and community formation. This, despite the fact that the scale of scientific problems demands broader, collaborative expertise, and with strong evidence suggesting that sharing can be beneficial. Consider the open arXiv preprint server, and open access PLoS journals which are examples of influential and successful scientific communities; and Steve Lawrence's recent article in *Nature* which shows strong correlation between open access and the number of citations [29].

The scientific method has evolved over centuries of practice and has the enviable feature that it is self-correcting, with committed and passionate practitioners. Thus despite these challenges, we are optimistic about the future of Open Science and the likelihood that its practice will be more open and collaborative than ever before. This optimism must also be tempered with the realization that changes of this magnitude can be generational, with new researchers quickly seeing the value of sharing if appropriate credit can be obtained when seeking career progression.

Bibliography

[1] Catalyst. http://catalyst.paraview.org/.

[2] CMake Usage Statistics. http://www.ohloh.net/languages/cmake.

[3] Code Review, Topic Branches and VTK. http://kitware.com/source/home/post/62.

[4] Coverity Scan: 2011 Open Source Integrity Report. http://softwareintegrity.coverity.com/coverity-scan-2011-open-source-integrity-report-registration.html.

[5] Directory of Open Access Journals. http://www.doaj.org/doaj?func=home&uiLanguage=en.

[6] Exchange Reproducibility Initiative. https://www.scienceexchange.com/reproducibility.

[7] History of the Royal Society. http://royalsociety.org/about-us/history.

[8] In science today, a genuis never works alone. http://www.guardian.co.uk/commentisfree/2013/feb/03/teamwork-science-transforming-the-world.

[9] Insight Segmentation and Registration Toolkit ITK. http://www.itk.org.

[10] MoleQueue HPC Resource Manager. http://wiki.openchemistry.org/MoleQueue.

[11] NA-MIC Publication Database. http://www.na-mic.org/publications.

[12] NIH Data Sharing Policy. http://grants.nih.gov/grants/policy/data_sharing/data_sharing_guidance.htm.

[13] Open Access Overview. http://www.earlham.edu/~peters/fos/overview.htm.

[14] Open Standards. http://en.wikipedia.org/wiki/Open_standard.

[15] OSA Interactive Science Publication. http://www.opticsinfobase.org/isp.cfm.

[16] ParaViewWeb. http://www.paraview.org/Wiki/ParaViewWeb.

[17] The CDash@Home Cloud . http://www.kitware.com/source/home/post/21.

[18] The Insight Journal. http://www.insight-journal.org/.

[19] The National Alliance of Medical Image Computing NA-MIC. http://na-mic.org.

[20] The Visualization Toolkit VTK. http://www.vtk.org.

[21] Wikipedia Editorial Oversight and Control. http://en.wikipedia.org/wiki/Wikipedia:Editorial_oversight_and_control.

[22] Xen virtualization platform. http://www.xen.org/.

[23] James Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. In C.D. Hansen and C.R. Johnson, editors, *Visualization Handbook*. Elsevier, 2004.

[24] Renee DiResta. Science as a service. http://radar.oreilly.com/2013/01/science-as-a-service.html.

[25] J. Gray. *The Fourth Paradigm: data-intensive scientific discovery*, chapter E-Science: a transformed scientific method. Microsoft Research, 2009.

[26] Capers J. Quality quest. *CIO*, February 1995.

[27] Jomier J., Jourdain S., Ayachit U., and Marion C. Remote visualization of large datasets with midas and paraviewweb. In *Web3D*, June 2011. http://www.kitware.com/publications/item/view/1285.

[28] Jomier J., Aylward S.R., Marion C., Lee J., and Styner M. A digital archiving system and distributed server-side processing of large datasets. *Proc. SPIE*, 7264, 2009.

[29] S. Lawrence. Free online availability substantially increases a paper's impact. http://www.nature.com/nature/debates/e-access/Articles/lawrence.html.

[30] K. Martin and W. Hoffman. *Mastering CMake*. Kitware, Inc., 2009.

[31] DOE Office of Science. The opportunities and challenges of exascale computing. http://science.energy.gov/~/media/ascr/ascac/pdf/reports/exascale_subcommittee_report.pdf, 2010.

[32] E. Ostrom. *Governing the Commons: The Evolution of Institutions for Collective Action*. 2009.

[33] B. Perens. Is opendocument an open standard? yes! http://www.dwheeler.com/essays/opendocument-open.html.

[34] K. Popper. *Conjectures and Refutations: The Growth of Scientific Knowledge*. 2nd edition, 2002.