Business Readiness Rating for Open Source

A Proposed Open Standard to Facilitate Assessment and Adoption of Open Source Software

Summary

Evaluating software is a critical task for corporate IT managers, but potential users of open source software lack an easy, effective, and trustworthy process for decision-making. There is no widely used model for evaluation. This complicates open source adoption, as companies assessing open source software can rarely learn from each other's experiences.

SpikeSource, the Center for Open Source Investigation at Carnegie Mellon West, and Intel Corporation have developed the Business Readiness Rating model, which lets IT managers quickly make informed and educated decisions about open source software. This model also allows users to feed their evaluations back into the open source community.

The calculation employed in the Business Readiness Rating model weights the factors that have proven to be most important for successful deployment of open source software in specific settings. Among these are functionality, quality, performance, support, community size, security, and others. The Business Readiness Rating model is open and flexible, yet standardized. This allows for broad implementation of a systematic and transparent assessment of both open source software and proprietary software.

Table of Contents

1 - The Challenge of Choosing Software

Deciding which software package to adopt inside an organization can be a challenging task. Along with the software's promised benefits come risks, such as issues of compatibility, usability, scalability, and even legality.

Traditionally, enterprises depended on commercial or proprietary software, despite limitations including:

- **Cost.** Generally high.
- **Closed source code.** The actual security and quality of the software is unknown.
- **Lock-in and lack of influence over roadmap.** The vendor chooses which improvements to make; customer requests may be ignored if they do not fit into the vendor's roadmap.

These factors offset the main benefit of using proprietary software:

- **Support.** Commercial software vendors have support staff dedicated to helping customers solve problems with the product.

Today, open source software is increasingly considered for business use. The reasons to do so include:

- **Cost.** It is often free.
- **Access to source code.** Since the source code is open, it benefits from automatic code review. Consequently, mature open source software projects tend to be more secure and have fewer bugs than their commercial counterparts.
- **Open architecture.** Open source software is often developed through a virtual community. Because the development community is often geographically distributed, open source projects are generally designed to be modular. Modular code is extensible and easy to debug.
- **Quality.** The source and architecture transparency described above enable well-managed projects to produce mature, high-quality products.

Despite these benefits, certain issues can hinder adoption. The number of open source projects is vast. Projects range from low-quality individual efforts to high-quality enterprise solutions. On SourceForge alone, over 100,000 open source projects are listed, with many more on other open source repositories like CodeHaus, Tigris, Java.net, ObjectWeb, and OpenSymphony.

Users and potential adopters of open source software face the following challenges:

- **Selection.** For some software categories, the choices are virtually limitless.
- **Support.** Most open source packages are not professionally supported.
- **Longevity.** Since most open source projects are not backed by commercial companies, the availability of future releases depends on community efforts.
- **Volatility.** Many open source projects follow the "Release Early, Release Often" paradigm to obtain traction in the community. Thus the only constant thing in the open source world is change. Many potential adopters of open source software are not ready to track and implement the rapid updates and changes of software packages prevalent in the open source world.
- **Low quality code for immature projects.** During conception, an open source project is usually developed by one or a few hobbyists or cash-strapped IT staffers who are passionate about creating something. These early developers may lack the resources or necessary experience to deliver a complete product. Projects may be orphaned rapidly once initial developers have created a

component that mostly solves their problem. New projects may be developed without formal software engineering or testing.

As a result, there is a broad continuum in the quality of open source software. Widely adopted open source projects frequently evolve into high-quality software products — often better than their commercial counterparts. However, immature open source software products may offer adopters more risks than benefits. There is a real need for a widely adopted, standardized method to assess the maturity of open source software. (In Appendix 2 of this paper, we list many of the characteristics of mature open source software.)

"It's fun until users ask you to support it." - Bill Joy on open source software development

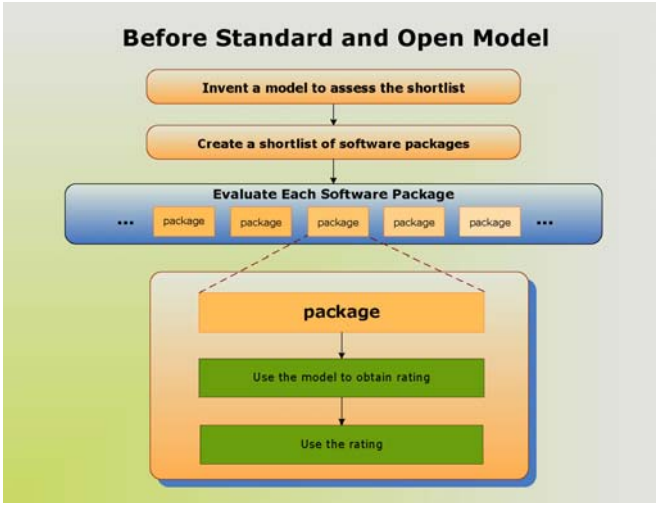
"Open source can sometimes defeat proprietary software. However, this is far from being always true." - Jean-Michel Dalle and Nicholas Jullien

"For all Open Source initiatives the first year is the largest hurdle." - CapGemini, OSMM

2 - Today's Software Assessment Practices

Deciding which software to deploy usually involves listing many options, and then doing a quick assessment to winnow them down into a shortlist that meets essential criteria. Creating a shortlist may be intimidating due to the plethora of open source software options available and the uncertainty about future versions. Yet without a shortlist, much time and energy will be wasted on evaluating unsuitable projects.

A good shortlist should include as many viable software options as possible while excluding those that aren't suitable. For certain application categories, software properties or standards may be useful filters, such as the application's primary language or the databases that it supports. However, for many application categories, such filters are hard to come by. Even after applying a simple filter like "programming language" to the 375 possible Content Management packages listed at http://www.cmsmatrix.org, the list will still be quite long.



After creating a shortlist, IT staff must more thoroughly evaluate the software packages that remain on the list. Most evaluators have to invent their own assessment methods, and without access to the assessment data or methods of their peers, they must re-assess each package themselves, even when others have evaluated the same software package.

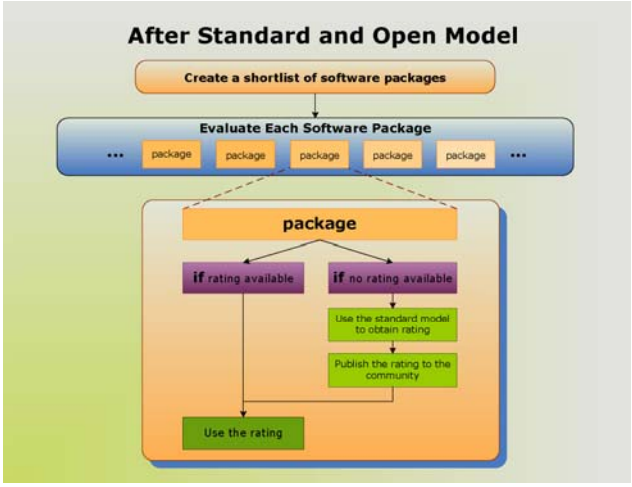
In practice, many software evaluation projects are done *ad hoc*, without a formal assessment methodology. Ad hoc methods may be incorrect or incomplete in their assessment, and it is extremely difficult to validate the correctness of the evaluation.

Inaccurate and incomplete evaluation mechanisms can lead to faulty decisions and product choices, which makes ad hoc assessment risky.

3 - An Open and Standard Model for Software Assessment

How can business users, developers, or IT engineers more easily decide which open source software to use? How can they confidently determine whether the software they are considering is mature and "business ready" enough for their purpose?

We propose that using an **open and standard** model to assess software will increase the ease and correctness of evaluation, and accelerate the adoption of open source software.



Additionally, an open and standard assessment model that is widely adopted and non-controversial would allow open source software users to share assessment results. Why *standard*? A standardized model allows common understanding of the assessment ratings. Why *open*? An open model promotes trust in the assessment process. It also ensures that the assessment model is flexible with respect to future changes. Validation of an open assessment model for correctness is straightforward: potential adopters of model will look at it, comment on it, and improve it.

Such a model should include the crucial requirements of a good software rating model — that it be Complete, Simple, Adaptable, and Consistent (CSAC):

- **COMPLETE**. The primary requirement for any product rating model is the ability of the model to highlight every prominent characteristic of the product, whether favorable or not. This is necessary so that the rating for any product is never misleading.
- **SIMPLE**. To gain wide acceptance, the model must be easily understood and relatively easy to use. Furthermore, the rating and terminology should be customer friendly. However, the model's completeness takes a higher priority.
- **ADAPTABLE**. Due to rapid changes in the software industry, any software rating model created today may be irrelevant in the future. During the conception stage, it is impossible to capture all future potential uses of the model. Therefore, we strive to build our model with adaptability in mind — and to keep it *open*. That way, when the model requires an extension, it will be easy to add one without much disruption of the current model.
- **CONSISTENT**. The scales and ratings that the model produces should be consistent across the model's different target uses. Comparable ratings for two software packages from two categories should signify equal business readiness.

Previous Models for Open Source Software Assessment

Different approaches exist to evaluate software. At least two previous initiatives aim to offer open source adopters a methodology for assessing and evaluating the suitability of open source software: the Open Source Maturity Model developed by Bernard Golden of Navicasoft, and publicized in his book *Succeeding with Open Source* (Addison Wesley, 2004); and the CapGemini Open Source Maturity Model, available from seriouslyopen.org. These models are both excellent pioneering efforts.

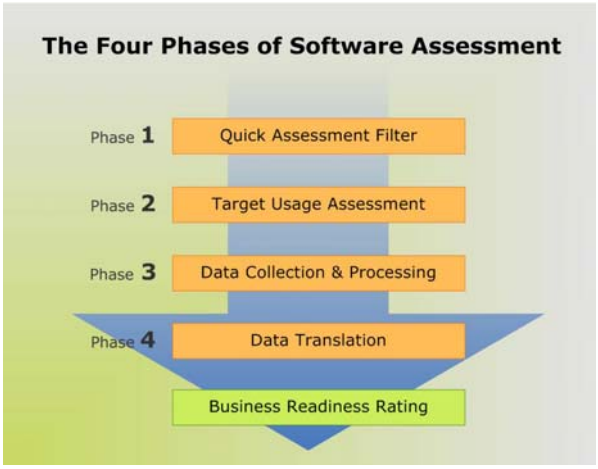
In proposing a new model, we use similar concepts for a thorough software evaluation, yet we provide more detailed evaluation data and scoring to assess the software's business readiness, especially for operational and support aspects. Separating particular areas of assessment and providing a weighted evaluation based on specific usage settings are some of the common concepts given by both Golden and CapGemini's models. We extend that concept and aim to provide a scientific model for such a rating system that contains a clear mapping from evaluation data to scoring, then the resulting final rating. In introducing our model, we wish to expand the idea of a methodology for software assessment into one that is widely adoptable and easily usable in as many evaluation situations as possible.

4 - Introducing the Business Readiness Rating Model

SpikeSource, the Center for Open Source Investigation at Carnegie Mellon West, and Intel Corporation want to simplify the assessment of open source software. To that end, we are jointly proposing a standard and open model: the ***Business Readiness Rating*** **(BRR)**.

This Business Readiness Rating model is intended to help IT managers assess which open source software would be most suitable for their needs. Open source users can also share their evaluation ratings with potential adopters, continuing the virtuous cycle and "architecture of participation" of open source.

In this model, we offer proposals for standardizing different types of evaluation data and grouping them into categories. To allow adoption of this assessment model for any usage requirements the software may have to meet, we separate the process of assessment into four phases.



First, a "quick assessment" to rule in or rule out software packages and create a shortlist of viable candidates. Second, ranking the importance of categories or metrics, third, processing the data, and last, translating the data into the Business Readiness Rating. A software component's Business Readiness Rating is scored from 1-5, with one being "Un-acceptable," and 5 being "Excellent."

In the sections below, we introduce the

Business Readiness Rating concept and a high-level overview of how to use the model. More extensive information on using the model can be found in Appendix 1.

Initial Filtering

To assess the business readiness of an open source software component, users may start by looking at several quantitative and qualitative properties of that component. During the initial **Quick Assessment phase**, a simple filter lets potential adopters quickly rule in or rule out software components with confidence.

We identified several viability indicators for use as filters in this phase, including:

- What is the licensing/legal situation of the software?
- Does it comply with standards?
- Are there referenceable adopters or users for it?
- Is a supporting or stable organization associated with the development efforts?
- What is its implementation language?
- Does it support internationalization and localization in your desired language?
- Are there third-party reviews of the software?
- Have books been published about the software?
- Is it being followed by industry analysts, such as Gartner or IDC?

Our list of filtering criteria for Quick Assessment is by no means exhaustive. Users may and should add filters that are important for the particular software package or situation they are evaluating. We include more elaborate Quick Assessment guidelines in Appendix 1.

Metrics and Categories

After completing the Quick Assessment process, it is important to look at which *metrics* and *categories* to use for the in-depth assessment phases.

We define a measurable property of an open source software project as a *metric.* Examples include: the number of books published about that software, the number of project committers, and the level of test activities. To create a standardized Business Readiness Rating, it is important to normalize the raw data of these metrics. Quantitative metrics, such as the number of downloads of a software package, are relatively easy to normalize. Qualitative metrics also need to be normalized. This process can be subjective, but we propose methods for quantifying and normalizing such metrics in Appendix 1 of this paper.

It is important to organize the assessment process into areas of interests, or assessment *categories*. We have defined 12 categories for assessing software:

Assessment Category	**Description**
Functionality	How well will the software meet the average user's requirements?
Usability	How good is the UI? How easy to use is the software for end-users? How easy is the software to install, configure, deploy, and maintain?
Quality	Of what quality are the design, the code, and the tests? How complete and error-free are they?

Assessment Category	Description
Security	How well does the software handle security issues? How secure is it?
Performance	How well does the software perform?
Scalability	How well does the software scale to a large environment?
Architecture	How well is the software architected? How modular, portable, flexible, extensible, open, and easy to integrate is it?
Support	How well is the software component supported?
Documentation	Of what quality is any documentation for the software?
Adoption	How well is the component adopted by community, market, and industry?
Community	How active and lively is the community for the software?
Professionalism	What is the level of the professionalism of the development process and of the project organization as a whole?

We define an assessment of software from a specific aspect as a *Category Rating.* A *category rating* is obtained by grouping together several metrics that measure the same aspects. How the rating in one category is calculated may differ from how another category is measured, but the results should use the same scale (1 to 5). One metric may contribute to several categories in different ways: for example, Fedora's release cycle of six months indicates a high level of community "liveness" but a low level of stability.

Tailoring for Functional Orientation

Functional orientation: the marriage of a type of component with its usage setting.

In our model, category ratings may be given different levels of importance depending on the software's usage requirements. Usage requirements can be derived from two factors:

1. From a *component type* (a collection of components that perform the same function, such as a Mail Transfer Agent (MTA), web container, web browser, office suite, etc. Users may interchange components within a component type without much loss or gain of functionality.
2. From a *usage setting.* The following are the usage settings for which software components tend to be assessed:

   Mission-critical use
   The software has to work 24x7. The company's business depends on it or it is part of a product/system offered by the company to its customers, who may depend on it. Examples: Apache, sendmail.

   Routine use
   The software is used internally. Workarounds or a short wait for updates are acceptable without having a major impact on the business.

Internal development and/or Independent Software Vendor

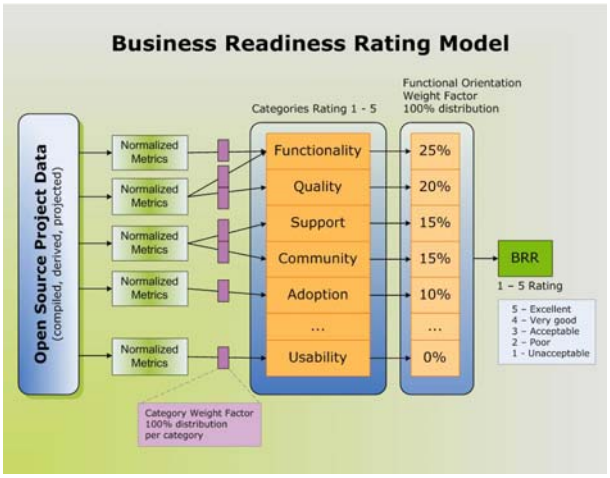
A development group may evaluate open source software to be integrated with internal systems for internal use, or for inclusion in a product or service for customers. The development group may be modifying the source code, with the expectation that they will be responsible for ongoing support.

Experimentation

Investigation of the software with no product-oriented objective in mind. Examples include comparative studies of similar products, exploration of modules of the software component to study implementation details, etc. The results of the research may lead to the software being used in one of the above roles.

The juxtaposition of a software package's *component type* (such as a web browser or a database) with its *usage setting* (such as advanced development and research, or mission-critical infrastructure) has significant impact on its business readiness. We define that combination of factors as the software's ***functional orientation***.

Although *category ratings* for areas such as documentation, code quality, or level of adoption can provide users with normalized and standardized numbers that are easy to understand, these numbers may mean different things for different *functional orientation* and usage requirements. An open source software component may have different levels of business readiness for different uses. One open source package may be considered business ready for ordinary use inside a company, but it may not yet be business ready for mission-critical utilizations.



The Business Readiness Rating for a software component is calculated by appropriately weighting a collection of category ratings according to the software's type and usage. Each functional orientation area focuses on only a limited number of category ratings; the Business Readiness Rating for the software's functional orientation is calculated based on those selected category ratings. This will ensure that the Business Readiness Rating reflects the most essential categories of the assessment. Best practices on how to select the category ratings most influential for any specific functional orientation, and how to choose the weighting factors, are described in Appendix 1.

Using the Model

The Quick Assessment phase and defining and ranking of *metrics* and *categories* according to their importance for the software's *functional orientation* leads us to the actions and steps taken in each assessment phase of the model to calculate the software's Business Readiness Rating.

Phase 1 – Quick Assessment

- Identify a list of components to be evaluated.
- Measure each component against the quick assessment criteria.
- Remove any components that do not satisfy user requirements from the list.

Phase 2 – Target usage assessment

Category weights

- Rank the 12 categories according to importance (1 – highest, 12 – lowest).
- Take the top 7 (or fewer) categories for that component, and assign a percentage of importance for each, totaling 100% over the chosen categories.

Metric weights

- For each metric within a category, rank the metric according to importance to business readiness.
- For each metric within a category, assign a percentage of importance, totaling 100% over all the metrics within one category.

Phase 3 – Data collection and processing

- Gather data for each metric used in each category rating, and calculate the applied weighting for each metric.

Phase 4 – Data translation

- Use category ratings and the functional orientation weighting factors to calculate the Business Readiness Rating score.
- Publish the software's Business Readiness Rating score.

5 - Conclusion

Our goal in proposing the Business Readiness Rating model is to offer a trusted and open framework for software evaluation. Our model aims to accelerate the software evaluation process with a systematic approach, facilitate the exchange of information between IT managers, result in better decisions, and increase confidence in high-quality open source software.

The Business Readiness Rating model is open and customizable, so that it can be applied to any business situation (it can even be used to assess proprietary software in a standardized way, by adapting some of the evaluation criteria). It is complete enough to allow IT staff from different companies and organizations to set their own assessment criteria, and perform their own full evaluations. By making the model open, we hope to guarantee its progressive evolution and improvement, and to increase its widespread adoption. Like other open source systems, the Business Readiness Rating model improves as its user base expands.

It is our goal to benefit the open source community by offering a vendor-neutral federated clearinghouse of quantifiable data on open source software packages to help drive their adoption and development. At http://www.openbrr.org, we will offer tutorials, discussion forums, samples, standard templates and worksheets of the Business Readiness Rating model for use by evaluators of open source software.

Appendix 1: Business Readiness Rating In-Depth

A software component's Business Readiness Rating is scored from 1 to 5, with one being "Unacceptable," and 5 being "Excellent." The "par" scoring, a rating of 3, or "Acceptable," passes crucial evaluation criteria, but may lag in areas that are desirable although not critical.

I. Best Practices for Quick Assessment

The goal of the quick assessment process is to quickly eliminate products that are clearly unsuitable, so that evaluators can focus their energy on assessing the most promising products. We identify the best practices to do so, which are the following:

1. **Decide the target usage of the application (Mission-Critical, Regular, Development, or Experimentation)**
   A component may or may not be a viable option, depending on how the application is intended to be used. An application that is semi-mature may be viable for experimentation use, but such an application may not qualify for regular or mission-critical use. Therefore it is pertinent for an evaluator to assess the intended usage of the application group (its *functional orientation)* before moving to any other steps.
2. **Select a handful of viability indicators based on target usage**
   We have identified several quick and easy-to-use indicators that strongly indicate the viability of a software package to be adopted. Our list is intentionally not comprehensive. Some of these viability indicators are so user-specific that trying to create an exhaustive list would be a fruitless exercise. Additionally, items on our list are not pertinent to all target usage. Evaluators should select only those few viability indicators that make most sense for their intended target usage. These indicators can include:

   Licensing/Legal

   Not all open licenses are created equal. Depending on the evaluator's target usage, some licenses are much more restrictive than others.
   Two open license properties are important for adopters of open source software to look at:
   1. Approved licenses. Open licenses are called approved licenses if they are recognized by the Open Source Initiative (OSI), http://www.opensource.org. These licenses have been thoroughly examined by OSI and are considered open and safe.
   2. Copyleft licenses. Copyleft licenses are open software licenses that allow modification of the code base and the redistribution of the modified version as long as the new product stays open. Copyleft licenses may pose restrictions for ISVs that are interested in integrating open source software into a commercial product. Visit http://www.fsf.org/licensing/licenses/ for more information about copyleft licenses.
   The common wisdom regarding open source licenses is:
   1. Regardless of the target usage, the license of the product should be one of the licenses recognized by Open Source Initiative (OSI), http://www.opensource.org.
   2. If you are an ISV, and looking for an open source application to

repackage as or integrate into a commercial software product, be aware of components that use copyleft licenses. Each copyleft license imposes different restrictions; familiarize yourself with the restrictions imposed on you before adopting copylefted components.

Standards compliance

For some software categories, compliance with standards is important, while for other categories standards may not exist. If compliance with industry standards is crucial for the software category being assessed, then we suggest that evaluators verify the compliance of the software under consideration before moving on to full BRR assessment.

Referenceable adopters

Some users are enthusiastic about being early adopters of new technologies/products. Others feel more comfortable being late adopters. If you belong to the second group, try to find references of other adopters and try to quantify your satisfaction with their findings.

Availability of a supporting or stable organization

If your organization does not have the resources to provide internal support, we suggest that you consider only applications for which professional support exists. The need for professional support may be ameliorated by the presence of a good supporting community.

Implementation language

The practice of adopting open source software often requires some customization work and internal support. It can be important to choose applications according to the coding expertise available in-house.

Third-Party Reviews

Adopters may want to investigate third-party reviews regarding the software to be considered.

Books

The availability of books about the software is a strong indicator of the software's level of maturity and adoption.

Followed by industry analysts, such as Gartner or IDC

Another strong indication for the viability of open source adoption is the availability of research reports on the software by analysts from leading market research firms like Gartner and IDC.

3. **Add more internal viability indicators to the list if applicable**
   Depending on adopters' specific situations, the new software package to be adopted may need to fulfill several critical requirements. These requirements should be included in the list of viability indicators for the quick assessment.
4. **Create a policy of the passing criteria**
   After compiling a list of viability indicators for a quick assessment, adopters need to create a policy of what assessment results are acceptable. Most of the viability criteria that we selected provide an affirmative or negative answer, analogous to red or green lights, perhaps with an intermediate yellow category. An acceptable policy might be "all green," or "no more than 2 yellows," etc.
5. **Assess each software component against the list of viability indicators**
   Having created a list of applicable viability filters and the passing policy, evaluators can start performing Quick Assessment for the software candidates. The result of the quick assessment determines whether or not the candidate software should go on to the full business readiness assessment process.

II. Best Practices for Target Usage Assessment

Although the model provides 12 assessment categories, it is not always wise to assess the software in all of them. Depending on the software's functional orientation, the relevance of certain assessment categories may not be significant enough for data-gathering efforts to be time-effective. Furthermore, the significance of the most crucial assessment categories may get diluted by the less important categories. To avoid this problem, we suggest that evaluators focus on no more than seven (7) assessment categories. This reduction in scope may also help evaluators in deciding the weighting contribution of the category ratings from each area. We suggest the following workflow to determine the weighting factors for each functional orientation:

1. Selecting the most important assessment categories.

During our exercises, we found that it is mentally difficult to eliminate assessment categories from consideration. It is natural for evaluators to expect good ratings from all assessment areas, and that leads to a hesitance to eliminate any categories. However, we believe that including all assessment categories is counterproductive. We suggest that evaluators:

- Rank the importance of the assessment categories from one to twelve (1-12) with respect to the software's functional orientation.
- Decide on a cut-off point in the ranked and sorted list of assessment categories, at the point where the difference in importance between one category and the following one is large (from "rather important" to "nice to have," for example). Evaluators should focus only on the ones that are above the cut-off point, and should discard/ignore the other categories. We suggest that evaluators focus on 7 or fewer assessment categories.

2. Selecting the appropriate weighting factors

Once evaluators have decided on the 7 or fewer assessment categories to focus on, the next step is to determine how much each category should contribute to the final result. These contribution levels are the *weighting factors,* and they are represented in percentages. All weighting factors together should add up to 100%. To determine a good distribution of weighting factors, assuming that you have 7 assessment categories, we suggest that evaluators:

- Sort the assessment categories to be used based on their ranking of importance derived in step 1 above.
- Assign an average weighting factor for the category that has the median ranking (middle of the sorted list).
- Work in a "zig-zag pattern" to assign weightings to the remaining categories. For example, if an evaluator focuses on 7 categories, the category ranked fourth would get an initial weighting of 15%. After that, the categories higher on the list (ranked first through third in importance), would be assigned weightings of 15% or more, and the categories lower on the list (ranked fifth through seventh in importance) would be assigned weightings no higher than 15%. A sample of a good weighting distribution for 7 categories can be: most important category: 25%, second-most important category: 20%, third-most important category: 15%, fourth-ranked/median category: 15%, fifth-ranked category: 10%, sixth-ranked category: 10%, and least-crucial category of the seven: 5%.
- Make sure that the sum of the weights is 100%
- Make final adjustments if necessary.

III. Best Practices for Data Collection and Processing

The data gathering and processing phase is by far the most time-consuming phase in the Business Readiness Rating model. Sufficient patience can be very rewarding though, as it can yield better data for assessment. We find that data is more easily obtainable for mature software packages. Therefore, the exercise can also be an initial indicator of the software quality.

Normalizing Metrics Measurements

All measurements within a category, whether they are qualitative or quantitative, need to be compared with a normalized scale that allows for the measurement to be meaningful.

For example, looking at the metric "downloads per month," if a software component has 2000 downloads per month, is that good or bad? How does that rate within a scale? A possible scale and scoring might be:

1 – 0 to 499 downloads/month – Unacceptable
2 – 500 to 999 downloads/month – Poor
3 - 1000 to 1999 downloads/month – Acceptable
4 – 2000 to 2999 downloads/month – Very good
5 – 5000 or more downloads/month – Excellent

If we looking at "total books published," a possible scale and scoring might be:

1 – 0 books – Unacceptable
2 – 1 to 2 books – Poor
3 – 3 to 5 books – Acceptable
4 – 6 to 15– Very good
5 – 15 or more books – Excellent

The metric is now normalized and can be used to compute a category rating.

To simplify the normalization for most metrics, whenever possible, we strive to be faithful to one general scale: a 1 to 5 numeric scale that is verbally translatable to Unacceptable, Poor, Acceptable, Very Good, and Excellent.

For quantitative metrics, mapping the raw numbers and score into a 1-5 scale should not be too difficult. For qualitative metrics, we realize that not all metrics can be measured in a range, or if they can, their range may not fit nicely into a scale of 5. For metrics that cannot be measured in a range, we suggest the following**:**

I. For binary (yes or no) metrics:

The negative answer to the metric should receive a 1 (unacceptable), and the affirmative one should receive a 3 (acceptable) *or* a 5 (excellent). The score for the affirmative response should be judged by how positive (how much better) it is compared to the negative one.

Example 1: "Security: is it monitored by CERT?" no = 1, yes = 5.
Any software package that is monitored by CERT has a much higher security quality than those that do not.

Example 2: "Security: is a security site/wiki available?" no = 1, yes = 3
A security site implies that the developers are putting some attention to the security quality of the software. However, the site alone is not enough. Therefore the affirmative answer is not much better than the negative one.

II. Some metrics are measurable in a range, but they cannot be measured on a scale of 5. For such metrics, we suggest assigning a score appropriate with the verbal scoring scheme above, and leaving the "spare scoring numbers" unused, or combining two or more answers/scores into one scoring number.

> **Example 1:** For the metric "Difficulty of entering the core developer team"
> - Anyone can contribute = 1 (unacceptable)
> - Rather difficult = 2 (poor)
> - Very hard = 5 (excellent)

We realize that the two scaling methods above may not work for all metrics. In fact, we suggest a radically different method to measure the metrics that contribute to a Functionality rating. However, we request that reviewers make an effort to try using these two scaling methods before liberally creating new ones. If a new scaling method needs to be devised, consider the consistency of the new scale compared with other metrics that affect the same category ratings.

Note that some of these metrics are imprecise. For example, the download metric may be compromised if the software is provided as part of some other software package — many open source components are included with several Linux distributions. The evaluation scoring may need adjusting to accommodate such a situation.

Processing Functionality Metrics

Functionality is an assessment category that is computed differently from other categories. Each type of software application has a unique set of features that needs to be fulfilled by the software package. Additionally some software packages may provide additional features that can be considered a plus point. Because of these differences, we are devising a unique method for processing and normalizing metrics that fall into the Functionality category to obtain the Functionality rating.

The Functionality rating is obtained by first comparing the features of the component being evaluated with a standard feature-set required for an average use. This standard feature-set must be constructed, or borrowed from an external source. For example, the feature set defined by CMS matrix (http://www.cmsmatrix.org) can provide the basis for evaluating content management systems. Once the standard feature-set is available, evaluators can begin the feature assessment using the following steps:

- Assign an importance score to all items in the feature list, using a scale of 1 to 3, with 1 being less important, 3 being very important.
- Compare the feature list of the component with the standard feature list. For each feature met, add the importance score to a cumulative sum. If not met, deduct importance score from the sum.
- If the software has extra features not on the standard feature list, assign an importance score for those features, and add their score to the sum.
- Divide the cumulative sum by the maximum score that can be obtained by the *standard* features only. This ratio is called the feature score. Using this scenario, it is possible to get a feature score that is higher than 100% or lower than 0%. This is intentional, as we want to "reward" extra features and "punish" for standard features that are missing.

- Normalize the feature score to a scale of 1 to 5 using this scheme:
  - less than 65%, score = 1 (unacceptable)
  - 65% - 80%, score = 2 (bad)
  - 80% - 90%, score = 3 (acceptable)
  - 90% - 96%, score = 4 (very good)
  - greater than 96%, score = 5 (excellent)

Category Rating Weighting Factors

Each metric within each category should have a weighting factor to differentiate the metric's importance within that particular category. For simplicity, the distribution of weighting factors for the metrics within a category should follow the same method as the functional orientation weighting factor (see "Selecting the appropriate weighting factors" in the "Ranking and Weighting Assessment Categories" section in this Appendix). Usually, since there are often fewer than 7 metrics contributing to a category rating, the weighting distributions for metrics can be decided more easily then for categories.

IV. Representative Metrics and their Scoring

NOTE: These metrics and scores are not the definitive set, but merely a representative set to illustrate the BRR model. We realize there are better candidates for metrics for measuring business readiness in each category. For many of them, the data is difficult to collect or simply unavailable. We provided an initial set to validate the usage of the model, and we will continue to improve the set of metrics in future versions of the Business Readiness Rating model.

Categories	***Metrics***	***Description***	***Scoring***				
			5 - Excellent	4 - Very good	3 - Acceptable	2 - Poor	1 - Unacceptable
Usability							
	End-user UI experience	This measures how well the UI is perceived by an end-user. (Intuitive interface/ navigation/control scheme)	Simple & Intuitive, information is well organized, no manual required		Takes little time to learn, information somewhat organized, some use of the manual		Complex, too much information, no obvious organization, cannot use without a manual
	Time for setup pre-requisites for installing open source software	The time/effort needed to set up a system, with all pre-requisites satisfied. This does not include OS.	< 10 minutes	10 - 30 minutes	30 min - 1 hour	1 - 4 hours	> 4 hours
	Time for vanilla installation/ configuration	Time it takes to get instant gratification, shows whether project is thinking about getting users up and running quickly as possible.	< 10 minutes	10 - 30 minutes	30 min - 1 hour	1 - 4 hours	> 4 hours
Quality							
	Number of minor releases in past 12 months	This measures planned updates and bug fixes. Typically, service packs in commercial products.	2		1 or 3		0 or > 3

Categories	Metrics	Description	Scoring				
			5 - Excellent	4 - Very good	3 - Acceptable	2 - Poor	1 - Unacceptable
	Number of point/patch releases in past 12 months	Typically, official point/patch releases are fixes for P1 bugs like deadlock, memory, and security vulnerabilities.	3 – 4		1 - 2, or 5 - 6		0 or > 6
	Number of open bugs for the last 6 months	This measures the quality of product usage.	< 50	50 - 100	100 - 500	500 - 1000	> 1000
	Number of bugs fixed in last 6 months (compared to # of bugs opened)	This measures how quickly bugs are fixed.	> 75%	60% - 75%	45% - 60%	25% - 45%	< 25%
	Number of P1/critical bugs opened	This measures the seriousness of quality issues found.	0	1 - 5	5 - 10	10 - 20	> 20
	Average bug age for P1 in last 6 months	This measures the responsiveness to fixing critical issues.	< 1 week	1 - 2 weeks	2 - 3 weeks	3 - 4 weeks	> 4 weeks
Security							
	Number of security vulnerabilities in the last 6 months that are moderately to extremely critical	This measures the quality related to security vulnerabilities. How susceptible the is software to security vulnerabilities.	0	1 - 2	3 - 4	5 - 6	> 6
	Number of security vulnerabilities still open (unpatched)	This measures whether the project is capable of resolving all security issues.	0	1	2	3 - 5	> 5
	Is there a dedicated information (web page, wiki, etc) for security?	This measures how aware of and seriously the project takes security issues.	Yes, well maintained		Yes		No
Performance							
	Performance Testing and Benchmark Reports available	This measures if there was any performance testing done and benchmarks published — typically in comparison to other equivalent solutions.	Yes, with good results		Yes		No
	Performance Tuning & Configuration	This measures if there is any documentation or tool to help fine-tune the component for performance. (Information about CPU, Disk, Network).	Yes, Extensive		Yes, Some		No
Scalability							
	Reference deployment	This measures whether the software is scalable and tested in real use through a real-world deployment.	Yes, with publication of user's size		Yes		No
	Designed for scalability	This measures whether component was designed with scalability in mind. Is it thread-safe? Does it run in a cluster environment? Can H/W solve performance problems?	Yes, extensive		Yes, some		No

Categories	Metrics	Description	Scoring				
			5 - Excellent	4 - Very good	3 - Acceptable	2 - Poor	1 - Unacceptable
Architecture							
	Are there any third-party plug-ins?	This measures the design for extensibility through third-party plug-ins.	> 10	6 - 10	2 - 5	1	0
	Public API / External Service	Allows for extensions via a public API, also shows design for customization.	Yes, extensive		Yes		No
Support							
	Average volume of general mailing list in the last 6 months	The general mailing list is the first place where people go for free help.	> 720 messages per month	300 - 720 msg per month	150 - 300 msg per month	30 - 150 msg per month	< 30 msg per month
	Quality of professional support	Professional support that helps fine-tune for the local deployment and troubleshooting is always desirable.	Installation + troubleshooting + integration / customization support		Installation support only		No professional support
Documentation							
	Existence of various kinds of documentation	A good documentation suite should include documentation for several user groups in several formats.	Install/deploy, user, admin, optimization, upgrading, development documentation is available in multiple formats (pdf, single html, multi-file html).	Install/ deploy, user, admin, upgrading guides available in several formats	Install/deploy and user guide available	Only text-based installation document ation exists	No proper documentation. A README file doesn't count
	User contribution framework	The best guides often come from user input / samples. This is feedback from people who have used the products.	People are allowed to contribute, and contributions are edited / filtered by experts		People are allowed to contribute		Users cannot contribute
Adoption							
	How many book titles does Amazon. com give for Power Search query: “subject:computer and title:component name”?	The availability of books is certainly good. A standard way of counting available books is important.	> 15	6 - 15	3 - 6	1 - 3	0
	Reference deployment	This measures whether the software is scalable and tested in real use through a real-world deployment.	Yes, with publication of user’s size		Yes		No
Community							
	Average volume of general mailing list in the last 6 months	The general mailing list is the place where the community helps itself.	> 720 messages per month	300 - 720 msg per month	150 - 300 msg per month	30 - 150 msg per month	< 30 msg per month
	Number of unique code contributors in the last 6 months	Code contributors usually promote the building of community around the project. The higher the number of code contributors, the better the community support.	> 50	20 – 50	10 - 20	5 - 10	< 5

Categories	*Metrics*	*Description*	*Scoring*				
			5 - Excellent	4 - Very good	3 - Acceptable	2 - Poor	1 - Unacceptable
Professionalism							
	Project Driver	The project driver performs project management, resource (money) gathering, etc.	Independent foundation supported by corporations (Apache / OSDL style)	Corporation (MySQL style)		Groups	Individuals
	Difficulty to enter core developer team	To ensure software quality, mature projects must be selective in accepting committers. New projects often have no choice.	Only after being active outside committer for a while		Rather difficult, must contribute accepted patches for some time		Anyone can enter

Appendix 2: Additional Information

I. The Characteristics of Mature Open Source Software

The following are typical characteristics of mature open source software. They apply in most situations, but not all of them.

1. Separation of development and stable branch.
2. The software is backed by a foundation, a corporation, or a strong community.
3. The community is organized into groups, each responsible for separate tasks (the maintainer, the documentation group, the development group, the evangelism group).
4. Project extensions are available.
5. The project has existed at least 1 year.
6. There is a well-defined process to enter the core development team.
7. The project's license is acknowledged by the Open Source Initiative (http://opensource.org).
8. Separation of documentations: User documentation, Installation documentation, Admin documentation, and, crucially, **development** documentation.
9. User documentation is extensive, available in many formats.
10. Separation of mailing lists: user mailing list, developer mailing list, security mailing list, evangelism mailing list, etc.
11. Not very aggressive in doing minor or major releases. Quick point releases are fine. (For Major.Minor.Point release numbering system.)
12. Books are readily available.
13. Large-scale adoption and usage of the software exists by organizations and/or individuals.
14. The software component has reasonable native unit and functional tests and the code coverage for these tests should be reasonable (30-80% range).
15. The component needs to be well integratable with other containing/contained components.
16. The component's bug database should indicate the revision numbers, unified diffs to each bug that is fixed.
17. The software is easy to install. It has well-documented installation instructions.
18. The software has a clean user interface. (GUI or command-line)
19. Performance metrics are available.
20. A deployment guide is available.
21. Well-known large-scale deployments (e.g. Wikipedia for mediawiki).
22. Intuitive to use and no convoluted designs (e.g., MoinMoin vs. Tikiwiki).
23. Ported across multiple platforms (Linux, Windows, Solaris, and Mac).
24. Non-intrusive, for example a small runtime footprint.
25. Separation of delivery of security patches, bug fixes, and new features/enhancements.

II. References

Top Tips for Selecting Open Source Software
http://www.oss-watch.ac.uk/resources/tips.xml

How to Evaluate Open Source Software / Free Software (OSS/FS) Programs
David A. Wheeler
http://www.dwheeler.com/oss_fs_eval.html

Choosing Open Source, A Guide to Civil Society Organizations
Mark Surman and Jason Diceman Jan 06 2004
http://www.commons.ca/articles/fulltext.shtml?x=335

Free and Open Source Software Overview and Preliminary Guidelines for the Government of Canada
http://www.tbs-sct.gc.ca/fap-paf/oss-ll/foss-llo/foss-llo17_e.asp

CapGemini's OSMM
http://www.seriouslyopen.org/nuke/html/index.php

Golden's OSMM
http://www.navicasoft.com/pages/osmmoverview.htm

Johnson, K.; A descriptive Model of Open-Source Software Development
http://sern.ucalgary.ca/students/theses/KimJohnson/toc.htm

Dalle, M.J., Jullien, N.; “Open Source vs. Proprietary Software”
http://opensource.mit.edu/papers/dalle2.pdf

CMS Matrix
http://www.cmsmatrix.org