The open source software "movement" has received enormous attention in the last several years. It is often characterized as a fundamentally new way to develop software that poses a serious challenge to the commercial software businesses that dominate most software markets today. The challenge is not the sort posed by a new competitor that operates according to the same rules but threatens to do it faster, better, cheaper. The OSS challenge is often described as much more fundamental, and goes to the basic motivations, economics, market structure, and philosophy of the institutions that develop, market, and use software.

"Open source" really does not describe a specific development process, but rather a set of legal and pragmatic arrangements designed to ensure the availability of source code. The arrangement allows a number of potential developers who are also users, to add or improve the functionality of the product. Their motivations are probably quite complex, but the open source arrangement provides some assurance that just as others receive the benefit of their work, they will receive the benefit of the work of others who also choose to work on the product.

A variety of different processes and development styles have evolved in different open source projects. Some, like the Apache server, have at their core a privileged group that decides what is committed to the code base. Others, such as Linux (until recently), have a single guru who is in control of all such decisions. Some projects are relatively "pure" open source, with all work contributed by volunteers. Others, such as the Mozilla project, are hybrids that include some full-time industrial staff in addition to large groups of volunteers.

Given the potential importance of open source, and the wide variety of open source styles of development, it is very important to understand how they compare, and how open source compares with more traditional industrial styles of development. The outcomes, such as code quality and response time to problem reports, are very important for obvious reasons. But given the absence of formal project planning and work assignments in open source, in order to understand how the code was developed and how the traditional roles of developer, tester, and bug fixer are filled by the core and peripheral members of the community. The problem, of course, is that traditional metrics (see, e.g., Carleton et. al, Fenton) are generally not available in open source projects. It is unrealistic to expect volunteer developers to report data on their activities in addition to contributing code and problem reports. Even when such metrics are reported, they may be unreliable (see, e.g., Herbsleb & Grinter), hence there are advantages to automatically collected data even for benchmarking of traditional industrial projects.

We have developed several measures that rely only on data sources that are automatically generated and generally available for open source and commercial projects. The main assumption of our work is that a major effect of a software technology (specifically various open source and commercial approaches) is to make it easier, simpler, or quicker for a developer to make certain modifications to a software entity whether to add a new functionality or to detect and fix a defect. The project measures are based on the analysis of changes to the software. First, we obtain a number of change measures (see, e.g, Mockus & Weiss), such as size, purpose, developer identity, and technology usage, from the change history of the source code that, in case of open source projects, is typically recorded in Concurrent Versioning System (CVS). These change measures together with information from problem report systems are then used to quantify the development process and the software product.

While directly collecting benchmarking measures is difficult and expensive in commercial setting (see, e.g., DeMarco), it becomes virtually impossible in an open source project. As we will see, the proposed process is largely automatic, inexpensive, non-intrusive, and applicable to most software projects using version control systems. Furthermore, it can be applied to an entire software project in its actual setting as we illustrate here to compare the Apache project to several commercial projects. We previously applied similar approach to evaluate the impact of a version-sensitive source code editor (Atkins et. al., 1999) and application engineering environment (Atkins et. al. 2000) on software change effort, and in a case study of OSS software (Mockus et. al.).

1.**Data Extraction**

The basic steps of the benchmark creation process are

- Investigate the development process and create a brief description of it. This is important in order to interpret the quantitative data in a meaningful way.

In order to produce an accurate description of the development process, our practice has been to write a draft based on published descriptions or with the help of a developer involved in the effort. This draft is then circulated among other core developers, who check it for accuracy and filled in missing details.

- Extract data about code changes.

We will refer to such changes as modification requests (MRs). Every change is recorded in CVS (or similar) system and contains the following tuple: date and time of the change, developer login, files touched, numbers of lines added and deleted for each file, and a short abstract describing the change. The abstract often needs further processing to identify people who submitted and/or reviewed the change and to obtain the Problem Report (PR) number for changes made as a result of a problem report.
• Extract data about problem reports.

Unlike CVS that is most commonly used in open source projects, there is more variety of problem reporting systems including bugdb and bugzilla. For each report, we extract the PR number, status (open, suspended, analyzed, feedback, closed), name of the submitter, date, and comment.

The data elements extracted from these archival sources are the used to construct a number of measures on each change to the code, and on each problem report. We used the process description as a basis to interpret those measures. Where possible, we then further validated the measures by comparing several operational definitions, and by checking our interpretations with project participants.

2. Open Source Benchmarks

Distribution of work

The contributors are ordered by the number of MRs from largest to smallest. The solid line in Figure 1 shows the cumulative proportion of changes against the number of contributors. The dotted and dashed lines show the cumulative proportion of added and deleted lines and the proportion of delta (an MR generates one delta for each of the files it changes). These measures capture various aspects of code contribution.

![Figure 1. The cumulative distribution of contributions to the code base.](image)

Participation in bug fixing can be measured by constructing a similar curve, selecting data only for those changes related to problem reports.

![Figure 2. Cumulative distribution of fixes.](image)

Productivity

It is probably not meaningful to try to measure productivity for all participants in open source project because there is likely to be a great deal of variation in the tail of the distribution. However, our case studies of Apache and Mozilla indicate that a relatively small group tends to contribute the vast majority of new functionality. Therefore, we measure the productivity of only this "core" group.

We defined "core" developers in the example below as groups of the most productive developers that contributed 83% of MRs (in the case of KMR/developer/year) and 88% of lines added (in the case of KLOC/developer/year). We chose these proportions because they were the proportions we observed empirically for the summed contributions of the 15 core Apache developers.

<table>
<thead>
<tr>
<th></th>
<th>Apache</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMR/developer/year</td>
<td>.11</td>
<td>.03</td>
<td>.03</td>
<td>.09</td>
<td>.02</td>
<td>.06</td>
</tr>
<tr>
<td>KLOC/developer/year</td>
<td>4.3</td>
<td>38.6</td>
<td>11.7</td>
<td>6.1</td>
<td>5.4</td>
<td>10</td>
</tr>
</tbody>
</table>

1. Defect density

We count as defects reported problems that resulted in actual changes to the code.

If we take a customer's point of view, we should be concerned primarily with defects visible to customers, i.e., post-release defects, and not build and testing problems. Most open source problems reports that we have encountered are very similar in this respect to counts of post-release defects, in that they were raised only against official, stable releases, not against interim development "releases."

However, if we are looking at defects as a measure of how well the development process functions, a slightly different comparison is in order, if we wish to compare open source to commercial projects. There is no provision for systematic system test in OSS generally. So the appropriate comparison would be to pre-system test commercial
software. Thus, the defect count would include all defects found during the system test stage or after.

The following table shows an example of both comparisons.

### Table 2. Comparison of defect density measures

<table>
<thead>
<tr>
<th>Measure</th>
<th>Apache</th>
<th>A</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post-release Defects/KLOCA</td>
<td>2.64</td>
<td>0.11</td>
<td>0.1</td>
<td>0.7</td>
<td>0.1</td>
</tr>
<tr>
<td>Post-release Defects/KDelta</td>
<td>40.8</td>
<td>4.3</td>
<td>14</td>
<td>28</td>
<td>10</td>
</tr>
<tr>
<td>Post-feature test Defects/KLOCA</td>
<td>2.64</td>
<td>*</td>
<td>5.7</td>
<td>6.0</td>
<td>6.9</td>
</tr>
<tr>
<td>Post-feature test Defects/KDelta</td>
<td>40.8</td>
<td>*</td>
<td>164</td>
<td>196</td>
<td>256</td>
</tr>
</tbody>
</table>

2. Time to resolve problems

In order to investigate time to resolve problems, we plot the cumulative distribution of PR interval, i.e., the elapsed time between PR open and PR closed. The distribution of PR resolution interval is approximated by its empirical distribution function that maps interval in days to proportion of PRs resolved within that interval. In the example below, Fifty percent of PRs are resolved within a day, 75% within 42 days, and 90% within 140 days.

![Cumulative distribution of PR resolution interval](image.png)

Figure 4. Proportion of PRs closed within given number of days.

3. Conclusion

While collecting benchmarking measures is difficult and expensive in commercial setting, it becomes virtually impossible in an open source project. Fortunately, automatically recorded data in change management systems can be analyzed to provide us with largely automatic, inexpensive, and non-intrusive benchmarks applicable to most software projects using version control systems. We hope that such an approach will enable basic comparisons of various types of projects and would highlight the inefficiencies in commonly used approaches, thereby suggesting ways to find improved ways to create software.

We are setting up a web site from which one can download the relevant scripts, and which shows benchmark data from a number of commercial and open source projects.