JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2016; **00**:1–22
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.1847

The Long-Term Growth Rate of Evolving Software: Empirical Results and Implications

Les Hatton,[1] Diomidis Spinellis,[2]*Michiel van Genuchten[3]

[1]*Kingston University, Kingston upon Thames, UK.*
[2]*Athens University of Economics and Business, Patision 76, GR-104 34 Athens, Greece.*
[3]*VitalHealth Software, Ede, The Netherlands.*

SUMMARY

The amount of code in evolving software-intensive systems appears to be growing relentlessly, affecting products and entire businesses. Objective figures quantifying the software code growth rate bounds in systems over a large time scale can be used as a reliable predictive basis for the size of software assets. We analyze a reference base of over 404 million lines of open source and closed software systems to provide accurate bounds on source code growth rates. We find that software source code in systems doubles about every 42 months on average, corresponding to a median compound annual growth rate (CAGR) of 1.21 ± 0.01. Software product and development managers can use our findings to bound estimates, to assess the trustworthiness of road maps, to recognise unsustainable growth, to judge the health of a software development project, and to predict a system's hardware footprint. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: code growth rate; software evolution; empirical study; compound annual growth rate; CAGR

1. INTRODUCTION

The amount of code in evolving software-intensive systems appears to be growing relentlessly. We believe that the main drivers for this phenomenon are two. First, users always demand ever more functionality and sophistication from computing systems. In addition, the exponentially increasing hardware's power [1, 2] accommodates and justifies corresponding investments in software. A NASA report lists some excellent examples of software code growth [3]: in a period of forty years, the percent of functionality provided by software to pilots of military aircraft has risen from 8% in 1960 (F-4 Phantom) to 80% in 2000 (F-22 Raptor). The F-4A had 1000 lines of software in 1960. The F-22A is reported to have 2.5 million lines of code. The F-35 will have 5.7 million lines of software.

Agreeing on a measure of growth rate for software however, turns out to be highly problematic. As we discuss in considerable detail later, the one factor that software growth does *not* exhibit in our

This is the accepted version of the following article: Les Hatton, Diomidis Spinellis, and Michiel van Genuchten. The long-term growth rate of evolving software: Empirical results and implications. *Journal of Software: Evolution and Process*, 2017, which has been published in final form at http://dx.doi.org/10.1002/smr.1847. This article may be used for non-commercial purposes in accordance with the Wiley Self-Archiving Policy http://olabout.wiley.com/WileyCDA/Section/id-828039.html.
*Correspondence to: Athens University of Economics and Business, Patision 76, GR-104 34 Athens, Greece. E-mail: dds@aueb.gr

studies is regularity. As a result we felt that a single parameter measure was the only thing that made real sense for the very disparate systems we analysed. Based on its pre-eminence in the financial world, where complex growth of equities and other assets matches the complexity we found in software systems, we chose the end-point CAGR, the compound annual growth rate over a period of time. This is defined as:-

$$\text{CAGR} = \left(\frac{S}{S_0}\right)^{1/r} \tag{1}$$

where S_0 stands for the software size at the beginning of the period, S stands for the size at the period's end, and r is the number of years in between. Other choices are possible such as the slope of the best fit linear curve on a log(S) v. time graph, but we felt that the end-point CAGR had the following advantages.

- It is easy to calculate.
- It is a familiar concept from the world of financial systems; using a concept familiar to general management may help them appreciate issues associated with software growth.
- In an important sense, since we chose only systems of more than one years's revision history and with a minimum line of code count of 50,000 lines, it averaged out some of the refactoring (whereby total source might be considerably reduced) which we found in abundance. Refactoring is of course important in an engineering sense but, almost by definition, it is not anticipated by developers and it is merely noise when we are trying to estimate the actual growth of a source code asset.

When applied to the avionic systems above, flight software had a CAGR of 1.18 over a period of 53 years. Robotic space flights had a CAGR of 1.17 between 1975 (Viking, 5000 lines of code) and 2005 (MRO, 545 thousands lines of code). Manned space flight showed a somewhat higher CAGR between the Apollo in 1968 (8500 lines of code) and the International Space Station (1.5 million lines of code) with a CAGR of 1.28 over this period. As we will see, these are consistent with the results extracted for the open source collection analysed here.

Originally, the growth of software was only a cost issue for senior management to consider: more developers needed to be hired or subcontracted. Increasingly, however, software determines more and more of a product's or service's user experience and becomes central to business as a sellable item, with the involvement of marketing and sales. The computer industry experienced this in the 1990s, the mobile phone industry went through the transition from a hardware-dominated to a software-dominated industry in the past decade, and sectors such as the car industry and the healthcare industry are likely to follow soon [4].

Parallels with Moore's law [1, 2], which states that integrated circuit component density doubles every 24 months, are sometimes drawn with software [5, 6, 7]. Interestingly, if software could indeed double every 24 months, then a 1,000 line program in 1965 (big but not huge then) would be around a 20 billion line program by now, very much bigger than any single program today. Software code is clearly growing more slowly than Moore's law but it has so far been unclear what that growth rate actually is. Determining the growth rate is highly relevant since software nowadays plays a major role in most domains of human activity [8, 9].

Based on the above, we limit the paper's scope to the establishment of the long-term statistical properties regarding the growth of evolving software. This goal purposefully excludes detailed qualitative or quantitative examination of topics such as: growing versus non-growing systems, phases of growth, types of growth, the correspondence between the type of a system and its growth, and the effect of implementation languages and platforms on growth. Although these are all worthy research topics, following them would detract us from our goal, which is to provide a single widely relevant figure regarding the compound aggregate growth of software code over a long time period.

This paper estimates software code growth rate based on an empirical analysis of some 404 million lines of code, from both open ($\approx$ 358 million lines from 1218 projects) and closed source products ($\approx$ 46 million lines from 9 projects). In the remainder of this paper we detail related work and empirical findings in the area of software growth (Section 2); the methods used to obtain the primary data, extract software code growth time series, and perform statistical analysis on these

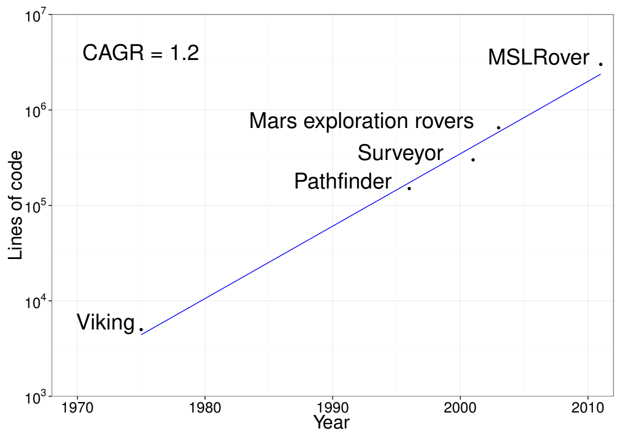

Figure 1. Software growth over the series of NASA Mars missions.

series (Section 3); the findings (Section 4), and the application and implications of this study's findings for the management of software (Section 6).

2. GROWTH OF SOFTWARE: A REVIEW OF THE LITERATURE

The evolution and growth of software has been the subject of decades of research [10]. A central theoretical underpinning regarding software's evolution are Lehman's eponymous laws as initially stated [11] and subsequently revised and extended [12, 13, 14]. This section will not expand on the topic, because a recent extensive survey of it [15] provides an excellent historical overview, and discusses the current state of the art. Along similar lines, a number of important studies focus on the stages of software growth [16, 17], as well as software aging [18] or decay [19]. Our focus instead is on work providing empirical evidence of software code growth. Many of the publications focus on open source software, which makes a natural candidate for studies on growth due to the availability of its source code. Fewer studies cover the growth of proprietary software, for data in that domain is scarce. Kemerer and Slaughter [20] discuss the difficulties associated with finding a good closed-source partner, who has the archives and data necessary, and who is willing to give researchers access to proprietary source code.

The lack of data regarding software code growth was one of the reasons given in the launch of a series of columns in *IEEE Software* in 2010 in which senior engineers and managers of diverse products were invited to describe the impact of software on their product as well as business. Companies that contributed a column were, among others, Honeywell, Hitachi, Microsoft, Oracle, Airbus, Shell Oil, Philips and Tomtom [21].

Many organizations provided the size of the product at different moments in time, which allows calculating a growth rate of the software over a period. An example of such a graph showing the growth of the software in Mars missions from 1973 to 2015 [30] is shown in Figure 1.

Table I contains an overview of the size and end-point CAGR of software products as they were described in the Impact columns as well as other published studies. The table shows the name of the product and its supplier in the first column. The second column indicates the period over which the growth was measured. Column 3 provides the period's duration in years, Columns 4 and 5 state the size at the beginning and end of the period, measured in millions of lines of code. (The significant digits listed depend on the precision of the available data in the corresponding reference.) Column 6 calculates the end-point calculated compound annual growth rate. The data set concerns over 59 million lines of code (MLOC). Comparing the growth rates of the products described allowed the calculation of a median compound annual growth rate of 1.16 with some variability quantified in a significance test later.

Table I. Compounded average growth rate in large software systems.

Product	Period	# Years	Start MLOC	End MLOC	CAGR	Reference
BIND (DNS implementation)	2000–2009	9	0.17	0.32	1.07	[22]
Oracle (Sun) Solaris	2000–2011	11	2.56	6.6	1.09	[23]
Honeywell flight management system	1978–2010	32	0.05	1.	1.10	[24]
Airbus A320	2006–2011	5	0.55	0.9	1.10	[25]
Airbus A380	2004–2008	4	1.8	2.7	1.11	[25]
Shell oil exploration software	1993–2012	19	0.3	2.2	1.11	[26]
Philips magnetic resonance scanner	2002–2010	8	3.5	9.5	1.13	[27]
Robotic space missions	1975–2005	30	0.005	0.55	1.17	[3]
Software in military aircraft	1960–2013	53	0.001	5.70	1.18	[3]
Fuji Xerox copier	1998–2008	10	2.	10.5	1.18	[28]
Tomtom navigation software	2004–2011	7	1.5	6.5	1.23	[29]
JPL Mars Rover	1982–2010	28	0.01	3.6	1.23	[30]
Manned space missions	1968–1989	21	0.009	1.50	1.28	[3]
Linux kernel	1994–2008	14	0.1	4.9	1.32	[31]
Samba (CIFS networking protocol)	1993–2009	16	0.006	1.05	1.39	[22]
Linux kernel	1994–1999	5	0.17	2.1	1.65	[32]
Total		272	12.73	59.6		

The following paragraphs discuss in more detail existing empirical work on software code growth. The selection of papers is based on their relevance to our study, namely their topic or the use of comparable methods or data. For each paper a summary in a few lines will be provided, as well as an overview of the empirical data that is presented in it.

Godfrey and Tu [32], in an early study of the Linux kernel's growth, showed that for several years its size was growing at a super-linear rate. The authors explained the fact by arguing that the linear growth of several subsystems can lead to the super-linear growth of their aggregation. A later study of the Linux kernel growth [31] investigated 800 versions of Linux over a period of 14 years. The authors found that "the average complexity per function, and the distribution of complexities of the different functions, are improving with time." The study contains size data for Linux from 1994 to 2008. The growth of the Linux kernel is also studied independently in this paper.

Details regarding the growth of software over several decades can be found in the NASA study on flight software complexity report [3]. According to this report, "In 2007 the NASA Office of Chief Engineer (OCE) commissioned a multi-center study to bring forth technical and managerial strategies to address risks associated with the growth in size and complexity of flight software (FSW) in NASA's space missions." Section 3 of the study's report (pp. 26–33) discusses software growth in flight software providing the figures presented in this paper's Introduction. The NASA study contains data that allows the calculation of CAGR for manned space, unmanned space, and software in military aircraft. The data are included in Table I.

Neamtiu and his colleagues [22] run an empirical study of the software evolution process over the lifetime of nine open-source projects. The analysis covered 705 official releases and a combined 108 years of evolution. The paper contains data that allows the calculation of CAGR for nine open source projects. The data associated with large systems are also included in Table I.

An in-depth study into the size and structure of the directories of the Arla system over 5 years and 62 releases [33] followed the growth of various measures over time. These measures included LOC (lines of code), SLOC (source lines of code—non-comment and non-blank), kB of object code, number of files and folders. The authors found a close similarity among growth patterns in terms of LOC, SLOC and kB. The Arla project grew from 40 kLOC (thousand lines of code) to 109 kLOC over 5 years giving a CAGR over this period of 1.22.

An earlier study [34] investigated whether open-source software systems grow more quickly than proprietary ones, but did not find evidence to support this hypothesis. The study found that the project growth is similar for all the projects in the analysis, indicating that other factors may limit growth. The data provided did not allow the calculation of the CAGR of the examined projects.

Capiluppi [10] tracked the size of nine open source projects over a period of time. The unit of measurement was kilobytes of object code. The author reported a constant growing size over time. The data provided was object code size, rather than LOC data that would allow the calculation of CAGR in the way it is defined in this paper.

Pei-Breivold and her colleagues [35] wrote a literature review on software evolution in open source systems. Papers up to 2009 were included and 41 were studied in detail. The main conclusions were that diverse metrics were being used, and few papers focused on the economic perspective. Eight out of the 41 studies provided the size in lines of code (LOC) in one shape or form. The paper does not present the LOC data that would allow the calculation of the software's CAGR.

A comparison between source LOCs and number of files as size metrics for software evolution analysis [36] concluded that the number of SLOCs is a metric as good as the number of files for software evolution analysis, based on a large sample of open source software. Examples of a non-sublinear growth pattern were given. Size is not provided at multiple moments in time, although authors calculate growth rates.

Koch [37] analyzed a total of over 6000 Sourceforge projects. Over 600 MLOC were added to the projects over the period investigated. According to the author, "the most interesting fact is that while in the mean the growth rate is linear or decreasing over time according to the laws of software evolution, a significant percentage of projects is able to sustain super-linear growth." The size data are not provided in the paper.

Unfortunately, the available empirical data were not enough to provide statistical robustness, so as a next step we examined the possibility of analyzing a large open source software corpus. Through its widespread adoption, open source software is affecting the software industry, science, engineering, research, teaching, the developing countries, and society at large [38]. A rather less obvious but no less important benefit of open source software is the availability of enormous amounts of source code and process data that can be used for study and experimentation. Including open source software in our study allowed us to expand the empirical base regarding software source code growth to 404 million lines of code.

3. METHODS

This paper's conceptual, operational, and quantitative model can be framed as follows [39]. The object of this study (**goal**) is to *characterize software code growth in a form that is useful to software product and development managers, in the context of medium to large software projects*.

Based on this goal the following **questions** are established.

Q1. How quickly does software code grow in terms of CAGR?
Q2. What is the variability and distribution of CAGR across diverse software systems?
Q3. Is it *realistic* to expect faster rates of growth? (This it not a technology question; it is a matter of statistical inference from existing practice.)
Q4. Do open and closed source systems grow at significantly different rates?

Finally, the **metrics** that can be used to answer these questions are the following.

M1. Project, Date, and lines of code time series (**Q1**)
M2. Compound annualized growth rate (**Q2**–**Q3**–**Q4**)
M3. 95% confidence intervals in the growth rate (**Q2**–**Q3**)

3.1. Reproducibility

The methods described in this section are provided in full along with the paper and the complete means to reproduce the results quoted, following published recommendations [40]. The raw data obtained from this study's repository analysis, the source code for the tools used to obtain and process it, and the R programs used for the statistical analysis are available online.[†]

3.2. Data Provenance

An overview of the systems studied appears in Table II.

Each row of the table contains the following details:

- a short name identifying the project (or set of projects);
- the source of the data in the form of a URL or a textual description (unless otherwise noted, under the HTTP or HTTPS protocol);
- the version control system used by the project: BZR for Bazaar [41], Git [42], *Hg* for Mercurial [43], RCS [44], or *svn* for Subversion [45];
- the number of projects falling under the umbrella of a larger project or organization;
- the number of time-series points examined (typically at a rate of one point per month); and
- the number of source code lines of code, in thousands.

The study contains data both from proprietary and open source software systems. The data from proprietary systems (those with the word "Impact" or a dash in the table's "data source" column) come mainly from systems examined in the *IEEE Software* "Impact" columns [46], as well as systems to which the authors had access through personal involvement. Open source systems were studied as a union of nine extra large projects listed in a paper on software growth [47], ten projects listed in a study of code churn [48], as well as well-known projects selected among popular "forked" repositories[‡] and repository "mirrors"[§] listed on the GitHub.com web site, those hosted on Launchpad source code hosting site,[¶] and those using the Mercurial version control system.[‖] Although the exhaustive rather than sampled nature of the sources used to select the open source software systems together with the very large amount of source code analysed do not guarantee a representative view of large open source software systems, they make a reasonable claim for it. Obviously, it cannot be claimed that the data concerning proprietary systems are comparably representative of their population in a statistical sense because the total amount analysed was so much smaller. However, it was still possible to perform an approximate significance test as described.

In order to examine a project's contents at successive time points, a local version of each project's repository was created, an exercise similar to an earlier one [49]. This allowed us to check out a version of the project at a specific time point, without incurring the delay and network bandwidth expense of fetching the code from a distant repository. Fortunately, all distributed version control systems [50] offer the ability to create a local mirror of a repository; this is an essential requirement for working without a central control point. Commands, such as `bzr branch`, `git clone`, and `hg clone` were used to create a complete local copy of each remotely-hosted repository. For projects using the Subversion system for version control, either used the *rsync* or the *svnsync* command and associated protocol were used to copy the remote repository's files into a local read-only instance of the remote repository. In total, over the three month long data collection period, 124GB of open source repository data were collected.

Some of the projects listed in Table II are umbrella projects, hosting many (e.g. 1,423 in the case of GNOME) independently developed sub-projects. These can be readily identified in the Table, because they have more than one project associated with them. The corresponding repositories were

[†] https://github.com/dspinellis/CAGR
[‡] https://github.com/repositories
[§] https://github.com/mirrors/
[¶] https://code.launchpad.net/projects
[‖] http://mercurial.selenic.com/wiki/ProjectsUsingMercurial

Table II. Examined projects

Project	Data Source	VCS	#prj	#points	kLOC
Apache	git.apache.org/	Git	562	47,749	106,420
Asterisk	lp:asterisk	bzr	1	196	1,072
Blender	lp:blender	bzr	1	160	2,360
C Python	hg.python.org/cpython	Hg	1	306	969
Django	github.com/django/django	Git	1	127	317
Drupal	git.drupal.org/project/drupal.git	Git	1	189	1,184
Eclipse	git.eclipse.org/c/	Git	780	57,246	116,147
GCC	github.com/mirrors/gcc.git	Git	1	327	8,452
GIMP	*	*	1	13	903
Git	github.com/git/git	Git	1	130	413
GitHub	github.com/search?o=desc&p=1&q=stars%3A%3E1&s=stars&type=Repositories	*	23	1,281	2,790
GNOME	git.gnome.org/browse/	Git	1,315	188,102	33,873
GNU	git.savannah.gnu.org/gitweb/	Git	787	86,259	41,271
GNU	rsync://svn.savannah.gnu.org/	svn	119	11,635	14,476
Gundalf	–	RCS	1	362	41
Inkscape	lp:inkscape	bzr	1	121	712
KDE	code.ohloh.net/file	Git	694	64,334	58,925
Linux	archive.org/details/git-history-of-linux	Git	1	293	6,225
Master	–	Git	1	248	420
MongoDB	github.com/mongodb/mongo	Git	1	100	3,073
Mozilla	github.com/mozilla/gecko-dev.git	Git	1	215	15,219
MySQL	lp:mysql-server	bzr	1	187	3,509
NetBeans	hg.netbeans.org/main	Hg	1	205	10,932
NodeJS	github.com/joyent/node	Git	1	84	3,732
Open Office	bitbucket.org/mst/ooo340	Hg	1	185	9,631
Perl	github.com/mirrors/perl.git	Git	1	338	935
PHP	github.com/php/php-src.git	Git	1	202	2,602
PostgreSQL	git://git.postgresql.org/git/postgresql.git	Git	1	235	1,252
R Project	lp:r-project	bzr	1	221	401
Ruby	lp:ruby	bzr	1	217	1,236
Ruby on Rails	github.com/rails/rails	Git	1	135	289
Safer C	–	RCS	1	27	83
Sourceware	sourceware.org/git/	Git	16	2,414	10,710
Spring	github.com/SpringSource/spring-framework	Git	1	91	986
Symfony	github.com/symfony/symfony	Git	1	73	291
Unix kernel	github.com/dspinellis/unix-history-repo.git	*	1	548	18,005
WildFly	github.com/wildfly/wildfly	Git	1	69	759
WINE	github.com/mirrors/wine.git	Git	1	272	3,706
Total			4,326	464,896	484,321

automatically identified on the project's web pages and then cloned through custom-written shell scripts, such as the one in Listing 1, which was used for cloning the Eclipse repositories. Note that the umbrella project as a whole was used during later statistical analysis however, for efficiency.

Listing 1: Code to download all Eclipse projects.

```
perl -e 'for ($i = 0; $i < 1000; $i += 50) {
 print "http://git.eclipse.org/c/?ofs=$i\n"
  }' |
while read url
do
  curl "$url" |
sed -n "/sublevel-repo/\
s|.*href='/c /\([^']*\)'.*|\
    git://git.eclipse.org/gitroot/\1|p" |
  xargs -n 1 git clone
done
```

Data for a few of the open source projects (marked with * in the "data source" column) were collected manually. Specifically, data for GIMP were collected by analyzing numerous snapshots of the project's source code distributions. A Git repository for Linux was constructed by cloning a historic version of the Linux kernel repository obtained from the internet Archive,** and then updating it with all recent changes using the `git pull` command. Finally, the data for the Unix kernel were collected by measuring the code through snapshots and version control data for a particular Unix evolution branch.

3.3. Repository Mining

Most modern projects and version control systems offer means to recreate a complete copy of a project's repository, which can then be used to obtain historical data with precision and accuracy. Extracting a time series of software size measurements from repositories requires a) a way to iterate over snapshots of the software at specific points of time, and, b) a method to measure the software size of that snapshot.

The start date used for each project was established by iterating over its snapshots and manually reading the project's commit log through its version control system. Each commit is associated with a date; the earliest one was typically the date that was used for starting the iteration. In some cases it was noted that an early commit marked an incomplete transition from an previous version control system or an era where no such system was in use. Such cases were characterized through rapid large addition of code in the project on some future date. In these cases the start date was adjusted to coincide with a date after all imported code was available within the version control system.

An iteration through a software system's snapshots measured a snapshot twelve times each year. It was decided to avoid measuring the code at the same date each month, for the irregularities of the Gregorian calendar [51] would cause the measurements to occur at varying intervals. Instead, $S = 365.25 \times 24 \times 60 \times 60$ was calculated as the number of seconds in a year, and $I = S/12$ was used as an increment to move from one measurement date to the next. All date calculations were performed by adopting the Unix convention of expressing time as seconds elapsed from 1970-01-01, and using the Unix *date* program for converting time points calculated in seconds into a Gregorian calendar date passed to the version control system.

Measuring the size of software is a difficult task, and the choice of the method can be controversial. It was decided to use as the metric physical lines [52] contained in source code files. A line is defined as a sequence ending with a line-feed character (ASCII LF—decimal character 10); this definition includes in the measure both executable statements and non-executable ones, such as comments and empty lines. The files containing source code (rather than compiled libraries, icons, documentation, translations, videos, and so on) were determined based on their extension (suffix).

**`http://archive.org/details/git-history-of-linux`

Table III. File Suffixes Used for Determining Source Code Files

Suffix(es)	Content
ada	Ada
as, asm, s, S	Assembly
c, C, cc, cpp, cxx, h, hh, hpp, hxx, i	C or C++
cs	C#
e	Eiffel
el, lsp	Lisp
F, F90	Fortran
g	ANTLR
go	Go
groovy	Groovy
idl	Interface description language
java	Java
js	JavaScript
lua	Lua
m	Matlab, GNU Octave, Objective C
m4	M4 macro
php, phpt	PHP
pl, pm	Perl
q	Apache Hive query
rb	Ruby
scala	Scala
sh	Unix Bourne shell
sql	SQL
vala	Vala
y	Yacc or Bison

The suffices used (see Table III) were determined by counting the number of lines of all text files and manually examining the entries ranked by line count at the top 2% range to see which suffixes corresponded to source code. The entry with the least number of lines comprised about 0.02% of the total text line count. The identification of whether a file was a text file or a binary one was performed by using the Unix *file* utility program. This utilizes several heuristics to determine a file's type, such as its first bytes (forming the so-called *magic number*) and the relative composition of characters within the file.

The measure used excludes binary and non-source code files. These are often third party libraries and tools included as compiled elements, files stored in a proprietary format, as well as graphic and sound resources. Some of them, such as icons or images used for a program's GUI, are clearly related to the effort required to develop the software and should be measured as part of it. Others, such as third party libraries, are irrelevant and should be excluded. However, in both cases the number of lines contained in a binary file is not an appropriate measure of its size, and for this reason it was decided to exclude binary files from the study's measurements. Non-source code text files were excluded, because these are often stored in formats, such as XML, where the line count metric is less relevant. Furthermore, many such files, such as input and output for regression tests, are generated by other tools, and therefore their size is not directly related to the software development effort.

Automatically-generated source code files are unlikely to affect the presented measurements. It is generally rare to include automatically-generated source code files in the version control repository. The rationale for this practice is to avoid tainting the project's revision history with massive auto-generated changes. Moreover, our measurement process did not involve the generation of automatically-generated source code files.

Based on the above, the shell script code for calculating the number of lines of a snapshot at a specific time point can be seen in Listing 2. In order to avoid the possibility of tainting the

Listing 2: Measuring lines of code.

```
echo -n "$YMD,"                # Date of time series
find $DIR -type f |            # Find all files
fgrep -v .git |                # Exclude VCS directory
# Include only source code files
egrep '\.(ada|as|asm|c|C|cc|cpp|cs|cxx|d|e|el|F|F90|g|go|groovy|
h|hh|hpp|hxx|i|idl|java|js|lsp|lua|m|m4|php|phpt|pl|pm|py|q|rb|
s|S|scala|sh|sql|vala|y)$' |
tr \\n \\0 |                   # Handle spaces in file names
xargs -0 cat |                 # Concatenate all files
wc -l                          # Measure their lines
```

measurement results between diverse version control systems or on varying execution platforms through accidental differences in the measurement code, the same measurement script supports through parameterization all the version control systems used by the code that was collected (Bazaar, Git, Mercurial, RCS, and Subversion), as well as its execution under Cygwin, FreeBSD, Linux, and Mac OS X.

3.4. Statistical Methods

Faced with such a large code base with which to experiment, the statistical analysis of the data started with exploratory data analysis to establish the basic properties of the underlying data set to understand the nature of growth. This was followed by a formal analysis based on robust statistical measures.

3.4.1. Exploratory Data Analysis The data were first aggregated into a comma-separated-values (CSV) file for each recognisable project with records in the format:-

```
YYYY-MM-DD, size in SLOC
```

After previously reported experience with a very small number of datasets [53], we had decided to characterise growth by using compound annualised growth rate (CAGR) using end-points. This is a non-trivial assumption and deserves some detailed justification.

Since it was not known what to expect initially, and to test the source code extraction, EDA techniques were used [54], simply to explore the data. Each of the 2,118 projects in the analysed data set was plotted both linearly and logarithmically and simply browsed to get a feel of how software code grows across this vast quantity of data. Figure 2 illustrates some of the extraordinary variety of observed types. These give fascinating insights into the progress of software projects in themselves given how simple they are to extract and display. The overall impression is that software growth data in source code repositories as studied here often defies simple categorisation.

There is already a significant amount of literature devoted to identifying and explaining linearity, sub- and super-linearity, ripples, rest periods and other more exotic growth types in software; see a published comparison [36] and the references listed in Section 2. However, there is no naturally compelling candidate model for growth as a whole, as becomes obvious when browsing through the many kinds of growth observed in this study. Indeed there is a bewildering array of bursts, retreats, oscillations, periods of inexplicable activity and the like. In association with this, software systems may or may not deploy re-use in its many forms at different stages of their life-cycle. Given this overall complexity, it seems obvious therefore to focus on the simplest possible form of growth, the end-point CAGR.

It turns out there are useful precedents that can be used from equally complex growth scenarios as occur for example in financial modelling. The general historic growth of equity values is accompanied by various kinds of complex change just as occurs in the growth of software projects. Even though there are many sophisticated auto-regressive models attempting to explain these growth

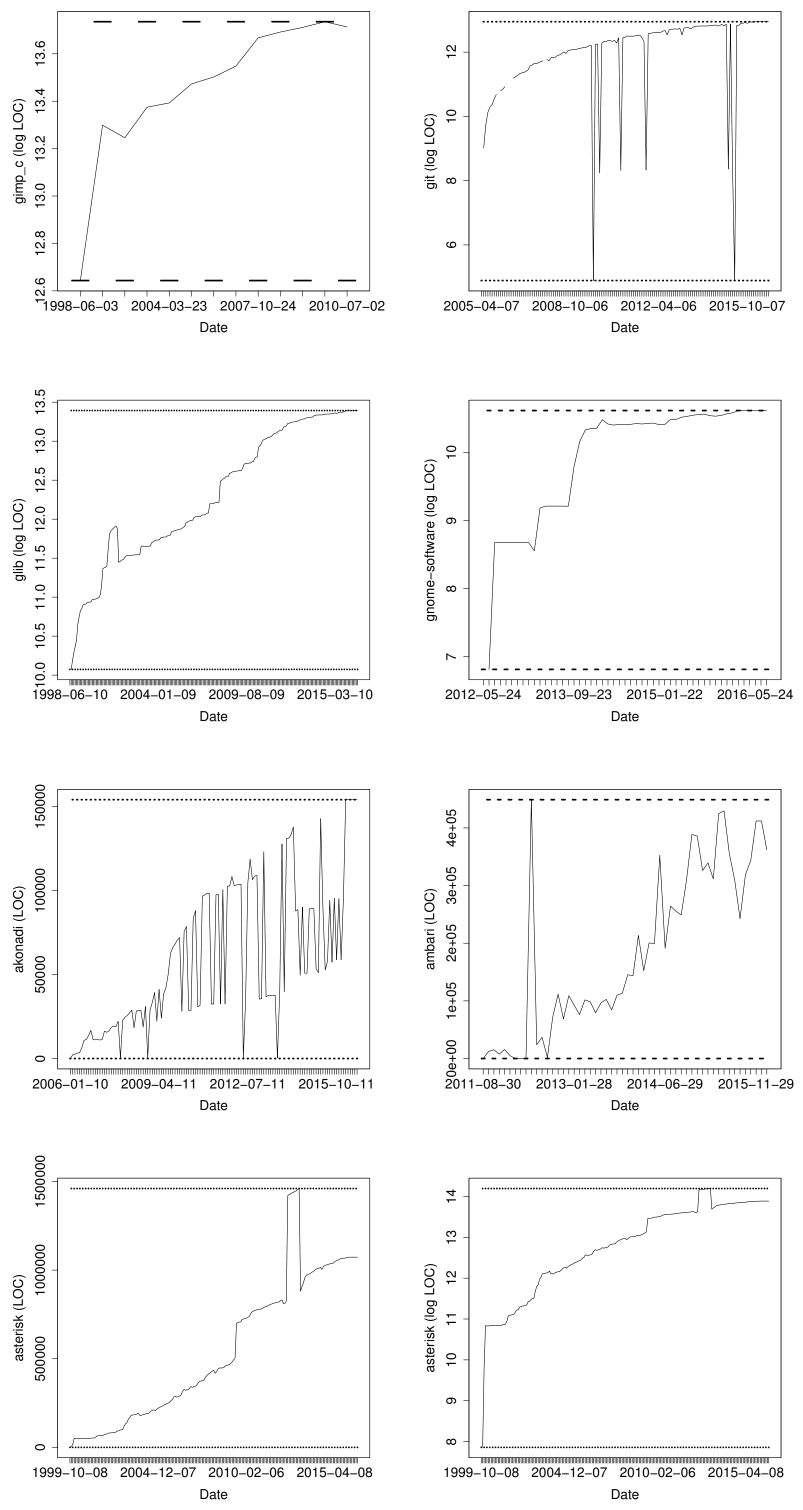

Figure 2. Some types of observed growth in both linear and logarithmic scales. Rows 1 and 2 show logarithmic growth of several projects; row 3 shows linear growth of two projects; and row 4 shows the same project in both linear and logarithmic scales.

Table IV. Size of open source datasets analysed treating umbrella projects as one.

Measurement	Number of lines of code	Number of projects
All projects	461837520	7465
Winnowed projects; empty (4452); small (0), large (0), short (555) or variable (339)	103623925	5346
Analysed projects	358213556	2118

patterns, one measure has emerged against which the performance of all financial algorithms and indeed the funds deploying them, tends to be used, the CAGR. Indeed, the growth of value is invariably couched in terms of percentage growth per annum simply because it is a very widely understood measure, even though it may at times bear little resemblance to the rapid day to day variations which actually occur.

It was therefore decided to adopt the same measure as a compact way of describing the observed growth of software systems in general without having to consider the detail of particular growth patterns for which there is already a substantial literature as quoted above.

As will be seen, the measure turns out to be extremely useful, and has led to a tight statistical distribution and consistency of growth across all of this very large dataset, allowing good estimates of both average growth and the expected degree of variation to be extracted. This justifies the value of CAGR as a global growth measure in software, just as it continues to be dominant in comparing the equally complex growth of financial instruments.

To model CAGR, the data were transformed into (r = YYYY-MM-DD, S = log(LOC)) pairs during analysis and the 0-100 percentile growth extracted, giving the annualised compound growth as a fraction. A system that exactly fits this appears as linear on a $\log S$ v. r plot. There is no reason to suppose that software systems would generally follow such a model closely, as it would require the exponentially increasing availability of new software developers. (Although, this can happen for substantial periods on some open source projects as seen below.) However, the model's averaged value is nevertheless very useful to calculate, and indeed departures from this model in real systems are often associated with well-known software engineering processes as will be seen.

3.4.2. Analysis The goal of this study is to investigate the growth rate of software as a CAGR *when its growth is formally and continuously integrated with a source control system over some significant part of its life-cycle*. Although somewhat subjective, the expectation of such a system is *continual* change and generally increasing functionality over some significant period, (nevertheless we allowed for a degree of refactoring).

Before analysing further, obviously wrong data were winnowed, for example, an empty project or if the time-line was non-monotonic. In addition, it was decided not to include those projects shorter than one year's duration in a source code control system since the measure extracted is annualised, and a minimum size of 50,000 lines of code was required.

Even after removing such obviously irrelevant projects, there remain numerous types of growth which do not meet our criterion of continuous and generally increasing growth over the entire life-cycle. For example, it was frequently observed that a huge amount of source code would suddenly be imported at some stage of the life-cycle and perhaps discarded again later. These appear to be characteristic of projects which were either trying out a particular revision control system and discontinuing, or finding a convenient place to store an effectively static repository, perhaps temporarily, or using the repository only sporadically as an archival facility, rather than an integral part of an evolving system. They do not seem to reflect the desired organic growth through active use of a repository over some significant period of time. Several examples are shown as Figure 3. Details of this *winnowing* process are shown in Table IV.

As can be seen from Table IV, discounting corrupted and empty projects, 72% of the projects were discarded (corresponding to 32% of the source code, since many projects were empty).

To understand the nature of the distribution of the resulting winnowed best linear fit growth data, a box plot with whiskers, histogram and quartile plot was generated using the standard R functions

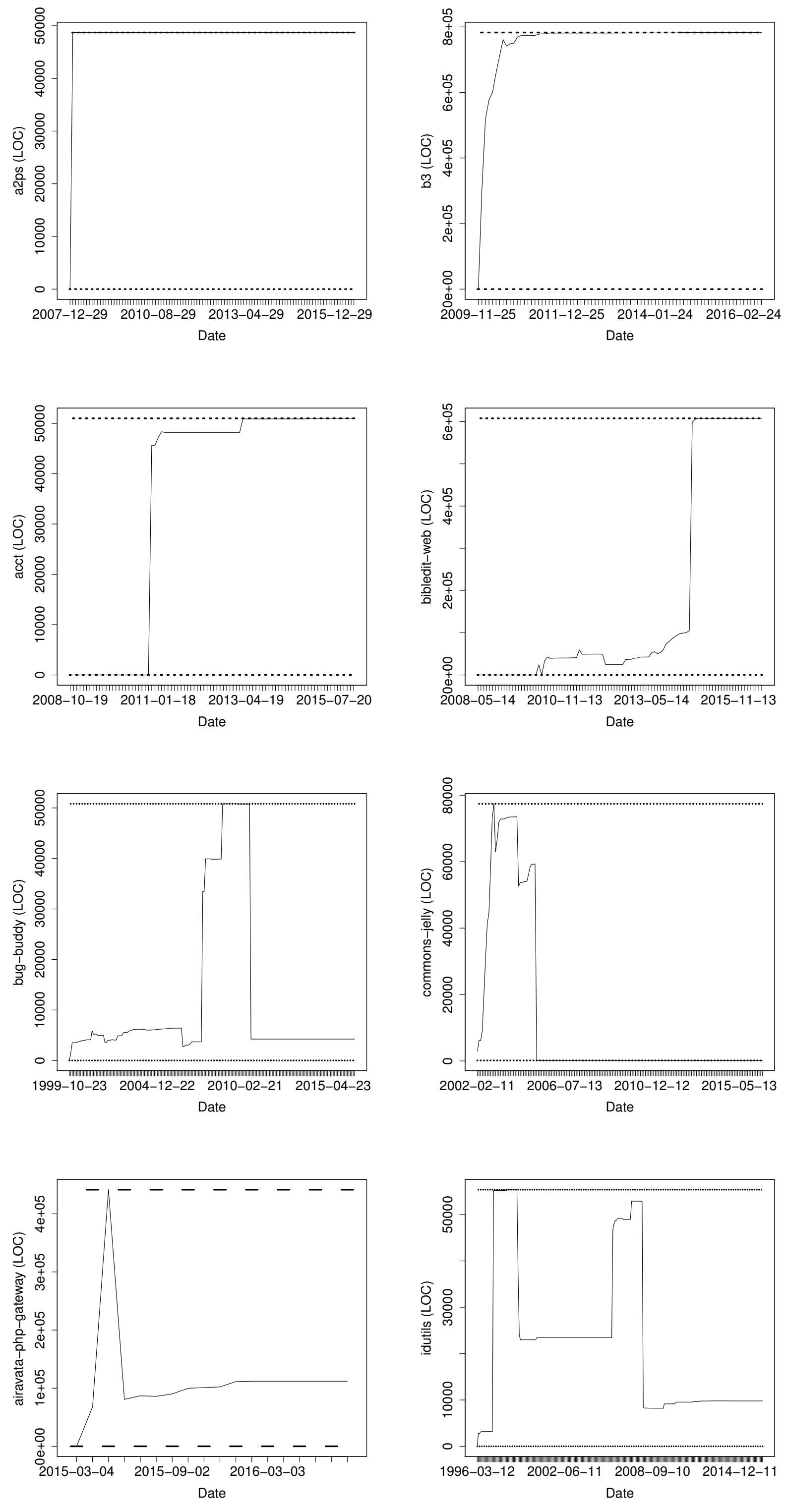

Figure 3. Examples of projects not considered to have normally evolving growth even though they would otherwise qualify as having a minimum of 50,000 lines of code and more than one year of activity. Projects like these (and there are many like them) are difficult to classify and appear to reflect considerable development activity outside of the repository, with occasional archiving into the repository.

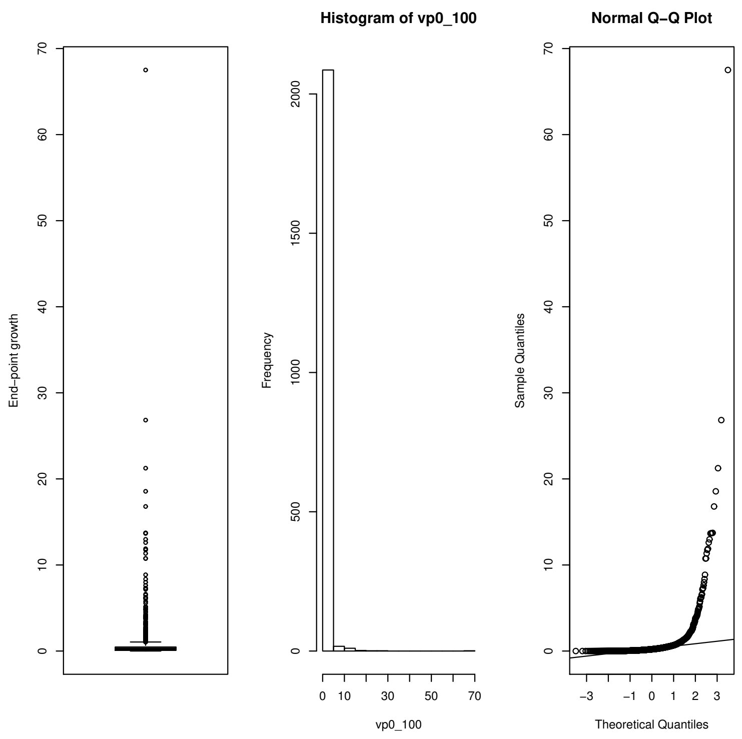

Figure 4. The distribution of end-point growths extracted from the winnowed data in box plot, histogram and quartile form.

Table V. Basic statistical properties of winnowed data.

Dataset	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Growths 0-100 percentile	0.00016	0.08101	0.20740	0.56630	0.46880	67.51000

boxplot(), hist(), qqnorm() and qqline() [54]. These are shown as Figure 4 and in tabular form as Table V for key parameters. The data are clearly right-skewed and long-tailed due to the impact of unusual growth or noise in the data and its effects on extracting the CAGR. In such contexts robust statistical measures are appropriate for estimates of location and scale [55] as, amongst other desirable properties, they are much less sensitive to the presence of the long tails.

Although not particularly efficient, two of the simplest robust estimators for location and scale are the median and the median absolute deviation (MAD) which exist conveniently in R as median() and mad(). These can be combined to give the following approximate (1-α) confidence interval for the mean μ of these data [56],

The median, $\hat{\mu}_{me}$, is a Fisher consistent estimator, resistant to gross errors and tolerant of up to 50% gross errors before it can be made arbitrarily large. The MAD is a simple robust scale estimator given by:-

$$\hat{\sigma}_{\mathrm{MAD}} = k \times \mathrm{med}(\mid x_i - \hat{\mu}_{me} \mid) \tag{2}$$

where $(x_i - \hat{\mu}_{me})$ denotes the i^{th} residual and k is a constant, (usually 1.4826 to make the estimator Fisher consistent for the normal model). The variance of $\hat{\mu}_{me}$ can be estimated by

$$\hat{V}(\hat{\mu}_{me}) = \frac{\pi}{2} \frac{(\hat{\sigma}^2_{\mathrm{MAD}})}{n} \tag{3}$$

in which case, an approximate (1-α) confidence interval for the mean μ is given by

Table VI. Basic statistical properties of project durations and sizes.

Dataset	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Durations in years	1.000	2.501	4.501	5.740	7.838	44.950
Size in LOC	40	5039	19910	164000	86630	18000000

Table VII. Estimated fractional growth including approximate 95% confidence intervals.

Method	95% lower	Median	95% upper
Growths 0-100 percentile	0.20	0.21	0.22

$$\hat{\mu}_{me} \pm z_{1-\alpha/2}\sqrt{\hat{V}(\hat{\mu}_{me})} \tag{4}$$

where $z_{1-\alpha/2}$ is the $(1-\alpha/2)$ quantile of the standard normal distribution.

These calculations are embedded in the R script analysis_growth.R in the deliverables package that accompanies this paper.

Finally, we analysed the distribution of project times and project sizes to give some idea of the duration of these projects. These are shown as Table VI

Like the growth data, these are significantly right-skewed. For completeness we decided to investigate if there was any correlation in the analysed dataset between the duration or size of a project and the CAGR of that project. It might be expected that perhaps only short lived projects would be able to sustain a high CAGR or there might be an unsuspected dependence on size. In fact, in the winnowed dataset, there are no such relationships. For project duration v. CAGR (R: lm(), Adjusted R-squared: 0.04221, $p < 2.2e^{-16}$) and for project size v. CAGR (R: lm(), Adjusted R-squared: -0.0003101, p = 0.5504). The duration data are shown as Figure 5 and the size data as Figure 6. Note that the size data is so right-skewed that we restricted attention to the vast majority of projects which contained less than 1 million lines of code. The absence of any obvious relationship in either simply emphasizes the extraordinarily diverse nature of growth in our dataset and supports our choice of a single-parameter estimate to characterise growth rather than a more nuanced model which could be in danger of fitting noise, just as CAGR has emerged to be the dominant measure of growth in the equally complex world of financial instruments.

As a final word on this analysis, perhaps the most valuable part of the exercise in retrospect was actually inspecting plots of the growth of all 2,118 projects as we did in the exploratory analysis phase. Although time consuming, the extraordinary diversity of what we found profoundly affected how we set about this analysis and how we might characterise the growth. This process is impossible to quantify or even describe adequately within the confines of journal publication, even as supplementary materials, so we would encourage interested readers to use the reproducibility package for this paper to have a look themselves.

4. FINDINGS

The following findings refer to the questions posed at the beginning of Section 3.

Q1. How fast does software code grow?

Using equation (4) gives estimates for the locations and 95% confidence intervals for the 0-100 percentile logarithmic growth in the winnowed data as shown in Table VII.

For the 2,118 data points extracted from the open source repositories, representing some 358 million lines of code, the compound annualised growth rate (CAGR) is found by adding 1 to the

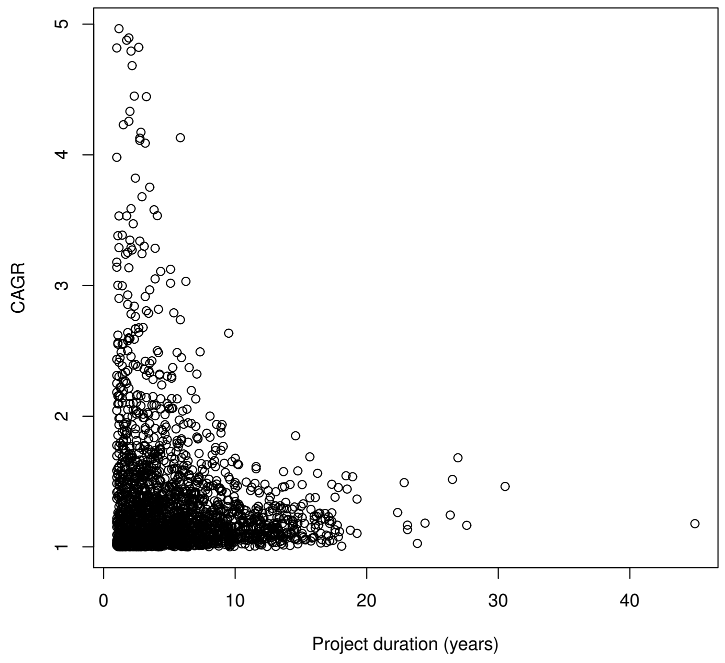

Figure 5. The project duration in years plotted against the extracted CAGR.

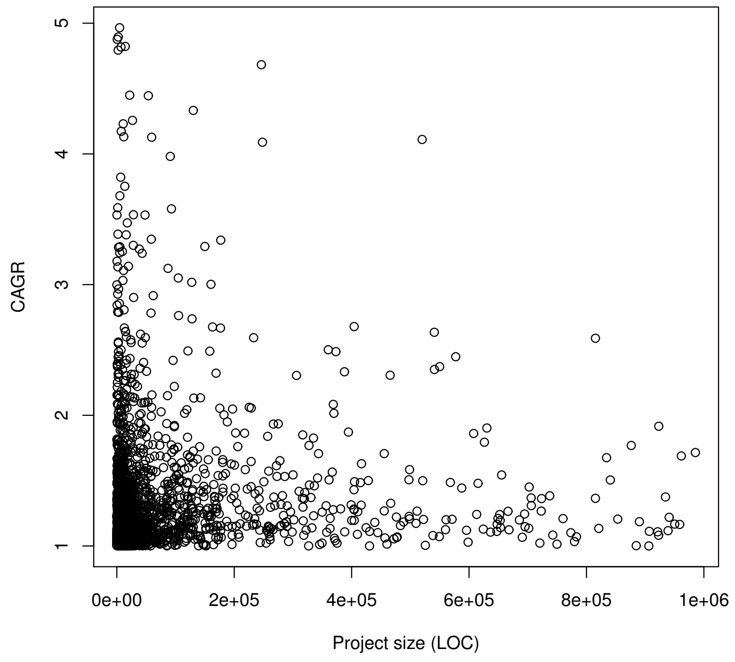

Figure 6. The project size in LOC plotted against the extracted CAGR.

fractional growths and, rounded to 2 decimal places is shown to be very likely in the range 1.20–1.22.

Q2. What is the variability and distribution of CAGR across diverse software systems?

Software growth is unquestionably more complicated than a simple CAGR model, exhibiting many forms of non-linear growth on either linear or logarithmic scales. For the 2,118 projects analysed, the estimated median CAGR 95% confidence interval was 1.21 ± 0.01 (0–100th percentile range). *(A CAGR of 1.21 means that software doubles every 42 months.)* There was no evidence to suggest any obvious relationship between either project duration in years and CAGR, or project size in LOC and CAGR. Instead extraordinary diversity of growth characterised our dataset.

Q3. Is it realistic to expect faster rates of growth?

The approximate 95% confidence interval for the CAGR is (1.20, 1.22). Optimistic software engineers and managers have always wanted to believe that with heroic efforts and great new tools, over-ambitious development deadlines can be somehow met, with frequent failures recorded as a result [57]. The studied dataset allows this belief to be reduced to simple statements in statistical inference. In essence, of all the many projects studied, very few of them achieved a CAGR much outside this range. It should be recalled that even though the distribution is right-skewed and long-tailed, it is unclear whether the very fastest growth rates are artificially induced by dumping large amounts of code from elsewhere into a repository or whether they genuinely represent rapidly growing development projects. We feel it is of more value to say that A has a .00001% chance of winning the lottery, rather than to say that A can win the lottery because B did.

Statistically, we can therefore say that the further outside these bounds an estimate might be, the less likely it is to be achieved over some substantial period. Moreover, as was explained earlier, exponentially increasing software size resulting from a constant CAGR implies the availability of exponentially increasing programming resources to fuel it, a situation which clearly can't continue ad infinitum. As economists say: "unsustainable trends are unsustainable" [5]. Furthermore, at some point it may be more economical to retire a system and invest in a new one [16, 17], rather than waste effort in the maintenance of aged code [19].

Q4. Do open and closed source systems grow at significantly different rates?

Numerous authors have included a discussion of both open and closed source systems in various contexts [58, 59]. This study adds to the literature by attempting to answer the question whether open and closed systems grow at the same rate.

Of course, there is a paucity of closed source data compared with open source but data for 9 large, closed systems was available, taken from Table I and totalling some 46 million lines of code, (about 13% of the total winnowed code base analysed). With some caveats, this is sufficient to allow a nonparametric test such as Wilcoxon-Mann-Whitney to be used in comparing the estimated location parameter for open and closed source.

In this case, the null hypothesis H0 is that the location parameters are the same for the open and closed datasets. The alternative hypothesis H1 is that they are different. Whether or not H0 is rejected depends on the p-value of the test. The two datasets were analysed using the wilcox.test() function of R for non-paired observations as embodied in the run_wilcox.sh script included in the accompanying deliverables package.

It was found that *there is no evidence to suggest that open source CAGR is significantly different from closed source CAGR.* (R: Wilcoxon-Mann-Whitney: W = 10142, p = 0.3354). This statement comes with a strong caveat that although the statistical test takes account of the very different sizes of dataset, it relies on each dataset being representative of the population from which it was sampled, which for the closed dataset is likely to be problematic at best. More data are essential to explore this.

5. THREATS TO VALIDITY

Beyond the caveats already mentioned in the narrative, it could be argued that analysing such large packages as Eclipse, GNOME, or KDE can bias the overall prediction. For example, KDE applications rely on the KDE desktop infrastructure and might therefore be expected to have growth factors in common. In fact, when the growth curves of each contributing package are analysed, they exhibit the same levels of variability as others in our population. Also, while KDE and GNOME form ecosystems, not all projects within their umbrella are tightly interlinked. Furthermore, other collections we have studied, namely GNU, Apache, and Sourceware, are simply agglomerations of projects that share common standards, support facilities, and values, rather than parts of a co-evolving ecosystem.

In some contexts, functionality that used to be delivered by one system, can in a next version be delivered by multiple linked systems (e.g. through external modules, web services, or a plugin-based architecture). Such changes will be reflected in our results as a lower CAGR estimate. Regarding the application of our results, given that a system's size S after r years of growth with a CAGR of p can be estimated from its current size as S_0 as $S = S_0 \times p^r$, the obtained size value is the same irrespective of whether S_0 and S are a single system or a conglomeration of modules.

The study contains data both from proprietary systems and open source ones. Although the number of proprietary systems in the study is small compared to the open source ones, this does not create a problem for the statistical methods we use. In addition, we do not attempt to perform a detailed comparative analysis on the differences between the two categories, just as we do not attempt to compare e.g. systems software with web-based applications. By applying a reasonable statistical test and giving our assumptions we conclude that there was no significant difference between the two categories, but that more data are necessary to reach deeper conclusions.

6. SOFTWARE MANAGEMENT APPLICATIONS AND IMPLICATIONS

The fact that the CAGR has been found to be within quite a narrow confidence interval could be used by software product and development management in several ways. We freely admit that the following possible applications are speculative as there are no economic data comparable to our main dataset of software growth but they are perceived as reasonable and useful by software managers and might encourage others to acquire such data.

6.1. Using CAGR to Bound Estimates

CAGR may be used to predict the development effort. Since the days of the COCOMO model [60], predicting the size of the software has always been a bottleneck to reliably estimate the software development effort and lead-time. Some have tried to use function points to arrive at better size estimates. One of the problems with estimation models has always been their limited "predictive powers" [61]. The suggestion advocated in this study is to use the project's CAGR. If the product is a million lines of code and its CAGR has been 1.20 for the last years, then it is likely that the organization will develop around 200 kLOC next year, assuming it does not drastically change its development team, process, or goals.

CAGR could also be applied to track the progress in large development projects. If the CAGR is calculated on a monthly basis (CMGR) one could track the growth of the product from month-to-month. Changes in the growth rate should probably be treated only as a trigger for further investigation by management, and this would work only in larger projects where the code base grows every month. This method will not be useful in projects that follow a traditional waterfall model where the team may spend weeks or months in a requirement or design phase without doing any coding.

6.2. *Assess the Realism of Road Maps*

It is possible to make size and effort estimates based on technology road maps. The semiconductor industry has famously planned its investments in new plants and processes based on Moore's law for decades [62, 63]. Similarly, the CAGR for software code growth may allow a manager to perform a reality check of a roadmap [64]. For example, if the roadmap features suggest that the software will grow for the next couple of years at a 30 percent rate, this is unlikely to be realistic. The argument that significant development resources can be added to make it happen anyway, can also be checked against previous attempts to do so, preferably within the same development environment. The response to such a roadmap from management should not be that 'this is impossible'. It is however valid to ask the question: 'please show me the data from within our company that indicates we may be able to do this'. It may be the case that the data are not available, for example because the company has not done similar software products for a long time. In that case one can try to look for similar size products in the data set this paper is based on and try to find similar growth rates to the one strived for. If those are not available either for a comparable product, it is unlikely that such an aggressive growth rate will be realised.

6.3. *Accelerated Growth May be Achieved for a Limited Time*

The data indicate that accelerated growth of the software can be achieved, but only for a limited time. An example in proprietary software was provided by Tomtom, which showed accelerated growth over a period of 5 years, followed by a period of slower growth [29]. In open source there are several examples of accelerated growth, with Linux as a prime example [31].

The implication for management is that it may be possible to catch up with a competitor by adding software resources to grow the software product faster. However, this may only work if it is possible to catch up in a limited period of time. And one should be aware that the market leaders typically invest more in the enhancement of their software product; they are not standing still. It remains true however that of the projects we analysed, very few achieved this. Fred Brooks [65] taught the software world that "adding manpower to a late software project makes it later". The data on growth of software may teach us that adding developers to a late roadmap will not allow the laggard to catch up with the market leader.

6.4. *Use CAGR as a Way to Predict a System's Hardware Footprint*

Managers could use CAGR to predict the size of a system in a few years. This may be helpful to estimate the required hardware footprint of a software-intensive product down the road. For example, the growing size of software source code implies the need for more RAM to run the executable code, more non-volatile storage to keep it, and higher bandwidth rates in order to receive software updates. This is relevant for products that are facing pressure on their hardware bill of materials, such as mobile phones, tablets, set-top boxes, car entertainment and navigation systems, and other high volume consumer devices. Thus through the use of CAGR, it becomes easier to predict what kind of platform the company will need to run its current applications three years into the future.

6.5. *Assess the Health of a Software Development Project*

The CAGR can also be tracked over time as an indicator of the health of an active software development project. A lower CAGR can have multiple causes. The first possible explanation is that software is not on the critical path. A second explanation can be limited investment. Limiting the development efforts will typically result in a lower CAGR [25]. A third explanation can be a refactoring phase that may lead to a lower CAGR for one or two years; after that the growth over new features and use cases will take over again [29]. A fourth explanation is of course a particularly difficult application area such as safety-related systems.

Thus far, the possible explanations are no reason for concern. This is not the case with the fifth explanation: a CAGR that gets lower over a long period could indicate a development department is getting stuck. It might be because a development process is temporarily (or even permanently) compromised by the disproportionate growth of defects, so that a large percentage of engineer resources is spent fire-fighting (corrective maintenance) rather than in the creation of reliable new functionality. Clearly, the product of a team with no or a few customer reported bugs per month will have more growth potential than a team that faces a flood of bugs every month. It would be interesting to have more empirical data on the relation between the defect density of a product in the field and its CAGR over time. Other possible explanations for getting stuck may be incorrect architectural choices or using an outdated platform.

The effort to do a CAGR-based assessment is very modest so the engineers should be asked to provide the amount of source code at various moments in time and calculate the CAGR over different periods. This should be no more than a few hours of work for an organization. A software code growth analysis cannot replace quality, regulatory, or process assessments, but it can provide a cheap, second opinion on the capabilities of the organization from a business perspective.

One of this work's authors has applied this approach in the assessment of a product's development over the years. The analysis showed that the CAGR went down from 1.25 in the initial years of the company to 1.06 after ten years. Further analysis indicated that the key engineers were more and more busy with fixing problems instead of adding new features. Interestingly, while the engineers were still questioning the validity of CAGR, the founder and CEO of the company took action to change the course by freeing up some key engineers for new developments and carefully evaluating which legacy products could be retired.

7. CONCLUSIONS

Using a simple CAGR model for the rate of growth on a very large population of mostly open source software, we found that source code in this population's systems grows by a factor of 1.21 ± 0.01 per annum, corresponding to a doubling of software size every 42 months.

Knowing that an evolving software asset is very likely to grow at a known fixed rate with reasonably tight 95% confidence bounds could be of great help in managing software projects. Future theoretical and empirical work can examine the reasons associated with the observed limits of CAGR and the time periods over which these limits can be exceeded. In addition, case studies can examine in depth how the software management uses of CAGR outlined in Section 6 can be applied in practice.

The software code's CAGR can be used to support development and test estimates and their funding impact, assess the trustworthiness of development roadmaps, predict a system's hardware footprint, assess the health of a development organization. Hopefully, software engineers, scientists, and managers will all benefit from new insights that can be built on the fact that there appear to be well defined bounds on the growth rate of this increasingly pervasive medium.

ACKNOWLEDGMENTS

The authors contributed equally to this work and their names appear in alphabetical order. The authors would like to thank Doug Vogel, Israel Herraiz, Panos Louridas, and Yiannis E. Spanos for insightful comments on earlier versions of this paper, as well as Dimitris Andreadis for providing access to historical JBoss repository data. The authors are also grateful to the paper's anonymous reviewers for their many constructive comments.

The work of the study's second author has been co-financed by the European Union (European Social Fund — ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) — Research Funding Program: Thalis — Athens University of Economics and Business — Software Engineering Research Platform as well as by the Research Centre of the Athens University of Economics and Business, under the Original Scientific Publications framework (project EP-2656-01).

REFERENCES

1. Moore GE. Cramming more components onto integrated circuits. *Electronics Magazine* 1965; **38**(8):114–117.
2. Mollick E. Establishing Moore's Law. *IEEE Annals of the History of Computing* Jul 2006; **28**(3):62–75, doi:10.1109/MAHC.2006.45.
3. Dvorak D. Final report: NASA study on flight software complexity. *Technical Report*, NASA, Office of the Chief Engineer 2009. URL http://www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf.
4. van Genuchten M. The impact of software growth on the electronics industry. *Computer* Jan 2007; **40**(1):106–108, doi:10.1109/MC.2007.36.
5. Humphrey W. The future of software engineering: Part I. *The Watts New? Collection: Columns by the SEI's Watts Humphrey*, Lynch R (ed.). Carnegie Mellon University, Software Engineering Institute: Pittsburgh, PA, USA, 2009; 63–68. URL http://www.sei.cmu.edu/library/assets/watts-new-compiled.pdf, originally published as news@sei, First Quarter 2001.
6. Rushby J. New challenges in certification for aircraft software. Current July 2013. Archived at WebCite as http://www.webcitation.org/6IRlxBt1R 2011. URL http://www.csl.sri.com/users/rushby/slides/emsoft11.pdf, computer Science Laboratory. SRI International. Menlo Park CA USA.
7. Lanza M. Software design & evolution. Lecture slides. Current July 2013. Archived at WebCite as http://www.webcitation.org/6IRmSG3qK 2011. URL http://www.inf.usi.ch/faculty/lanza/Education/SDE-2011/Lec02Maintenance.pdf, faculty of Informatics. Universitá della Svizzera italiana.
8. Aho A. Software and the future of programming languages. *Science* 2004; **303**(5662):1331–1333, doi:10.1126/science.1096169.
9. Joppa L, McInerny G, Harper R, Salido L, Takeda K, K O, Gavaghan D, Emmott S. Troubling trends in scientific software use. *Science* 2013; **340**:814–815.
10. Capiluppi A. Models for the evolution of OS projects. *ICSM '03: International Conference on Software Maintenance*, 2003; 65–74, doi:10.1109/ICSM.2003.1235407.
11. Lehman M. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software* 1979; **1**:213–221, doi:http://dx.doi.org/10.1016/0164-1212(79)90022-0.
12. Lehman MM, Belady LA. *Program Evolution: Processes of Software Change*. Academic Press, 1985.
13. Lehman MM, Ramil JF, Wernick PD, Perry DE, Turski WM. Metrics and laws of software evolution — the nineties view. *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*, IEEE Computer Society: Washington, DC, USA, 1997; 20–32.
14. Lehman MM, Ramil JF. Rules and tools for software evolution planning and management. *Annals of Software Engineering* 2001; **11**(1):15–44, doi:10.1023/A:1012535017876. URL http://dx.doi.org/10.1023/A%3A1012535017876.
15. Herraiz I, Rodriguez D, Robles G, González-Barahona JM. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Surveys* Nov 2013; **46**(2):1–28, doi:10.1145/2543581.2543595.
16. Lehner F. Software life cycle management based on a phase distinction method. *Microprocessing and Microprogramming* 1991; **32**(1OCo5):603 – 608, doi:http://dx.doi.org/10.1016/0165-6074(91)90409-M. URL http://www.sciencedirect.com/science/article/pii/016560749190409M, euromicro symposium on microprocessing and microprogramming.
17. Bennett KH, Rajlich VT. Software maintenance and evolution: A roadmap. *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, ACM: New York, NY, USA, 2000; 73–87, doi:10.1145/336512.336534.
18. Parnas DL. Software aging. *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, IEEE Computer Society Press: Los Alamitos, CA, USA, 1994; 279–287. URL http://dl.acm.org/citation.cfm?id=257734.257788.
19. Eick SG, Graves TL, Karr AF, Marron JS, Mockus A. Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng.* Jan 2001; **27**(1):1–12, doi:10.1109/32.895984. URL http://dx.doi.org/10.1109/32.895984.
20. Kemerer CF, Slaughter S. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering* 1999; **25**(4):493–509.
21. van Genuchten M, Hatton L. Metrics with impact. *IEEE Software* Jul/Aug 2013; **30**(4):99–101, doi:10.1109/MS.2013.81.
22. Neamtiu I, Xie G, Chen J. Towards a better understanding of software evolution: an empirical study on open-source software. *Journal of Software: Evolution and Process* 2013; **25**(3):193–218, doi:10.1002/smr.564.
23. King C, Beal C. CSI kernel: Finding a needle in a multiterabyte haystack. *IEEE Software* Nov/Dec 2012; **29**(6):9–12, doi:10.1109/MS.2012.154.
24. Avery D. The evolution of flight management systems. *IEEE Software* Jan/Feb 2011; **28**(1):11–13, doi:10.1109/MS.2011.17.
25. Burger S, Hummel O, Heinisch M. Airbus cabin software. *IEEE Software* Jan/Feb 2013; **30**(1):21–25, doi:10.1109/MS.2013.2.
26. van Malkenhorst M, Mollinger L. Going underground. *IEEE Software* May/Jun 2012; **29**(3):17–29, doi:10.1109/MS.2012.62.
27. Hofland L, van der Linden J. Software in MRI scanners. *IEEE Software* Jul/Aug 2010; **27**(4):87–89, doi:10.1109/MS.2010.106.
28. Tsuchitoi Y, Sugiura H. 10 MLOC in your office copier. *IEEE Software* Nov/Dec 2011; **28**(6):93–95, doi:10.1109/MS.2011.133.
29. Schaminée H, Aerts H. Short and winding road: Software in car navigation systems. *IEEE Software* Jul/Aug 2011; **28**(4):19–21, doi:10.1109/MS.2011.83.
30. Holzmann G. Landing a spacecraft on Mars. *IEEE Software* Mar/Apr 2013; **30**(1):83–86, doi:10.1109/MS.2013.32.

31. Israeli A, Feitelson DG. The Linux kernel as a case study in software evolution. *Journal of Systems and Software* Mar 2010; **83**(3):485–501, doi:10.1016/j.jss.2009.09.042.
32. Godfrey M, Tu Q. Evolution in open source software: a case study. *International Conference on Software Maintenance*, 2000; 131–142, doi:10.1109/ICSM.2000.883030.
33. Capiluppi A, Morisio M, Ramil J. Structural evolution of an open source system: A case study. *12th IEEE International Workshop on Program Comprehension*, 2004; 172–182, doi:10.1109/WPC.2004.1311059.
34. Paulson J, Succi G, Eberlein A. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering* Apr 2004; **30**(4):246–256, doi:10.1109/TSE.2004.1274044.
35. Breivold H, Chauhan M, Babar M. A systematic review of studies of open source software evolution. *17th Asia Pacific Software Engineering Conference (APSEC)*, 2010; 356–365, doi:10.1109/APSEC.2010.48.
36. Herraiz I, Robles G, González-Barahona JM. Comparison between SLOCs and number of files as size metrics for software evolution analysis. *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, IEEE Computer Society: Washington, DC, USA, 2006; 206–213.
37. Koch S. Software evolution in open source projects—a large-scale investigation. *Journal of Software Maintenance and Evolution: Research and Practice* Nov 2007; **19**(6):361–382, doi:10.1002/smr.348.
38. Androutsellis-Theotokis S, Spinellis D, Kechagia M, Gousios G. Open source software: A survey from 10,000 feet. *Foundations and Trends in Technology, Information and Operations Management* 2011; **4**(3–4):187–347, doi: 10.1561/0200000026.
39. Basili V, Caldiera C, Rombach DH. Goal question metric paradigm. *Encyclopedia of Software Engineering*, vol. 2. John Wiley and Sons: New York, 1994; 528–532.
40. Ince D, Hatton L, Graham-Cumming J. The case for open program code. *Nature* February 2012; **482**:485–488, doi:10.1038/nature10836.
41. Gyerik J. *Bazaar Version Control*. Packt Publishing Ltd: Birmingham, UK, 2013. ISBN 978-1849513562.
42. Loeliger J, McCullough M. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc.: Sebastopol, CA, 2012. ISBN 978-1449316389.
43. O'Sullivan B. *Mercurial: The definitive guide*. O'Reilly Media, Inc.: Sebastopol, CA, 2009. ISBN 978-0596800673.
44. Tichy WF. Design, implementation, and evaluation of a revision control system. *Proceedings of the 6th International Conference on Software Engineering*, ICSE '82, IEEE, 1982; 58–67.
45. Pilato CM, Collins-Sussman B, Fitzpatrick BW. *Version control with Subversion*. O'Reilly Media, Inc.: Sebastopol, CA, 2009. ISBN 978-0-596-51033-6.
46. van Genuchten M, Hatton L. Software: What's in it and what's it in? *IEEE Software* Jan/Feb 2010; **27**(1):14–16, doi:10.1109/MS.2010.19.
47. Ingo H. How to grow your open source project 10x and revenues 5x. Current July 2013. Archived at WebCite as `http://www.webcitation.org/6ITPs7RJW` 2010. URL `http://openlife.cc/blogs/2010/november/how-grow-your-open-source-project-10x-and-revenues-5x`.
48. Kraaijeveld J. Mining git repositories and understanding code churn. Current July 2013. Archived at WebCite as `http://www.webcitation.org/6ITQFgcbD` 2013. URL `http://kaidence.org/posts/mining-git-repositories-and-understanding-code-churn.html`.
49. Mockus A. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, IEEE Computer Society: Washington, DC, USA, 2009; 11–20, doi:10.1109/MSR.2009.5069476.
50. O'Sullivan B. Making sense of revision-control systems. *Communications of the ACM* Sep 2009; **52**(9):56–62, doi:10.1145/1562164.1562183.
51. Dershowitz N, Reingold EM. *Calendrical Calculations*. Cambridge University Press: Cambridge, 1997.
52. Kan SH. *Metrics and Models in Software Quality Engineering*. second edn., Addison-Wesley: Boston, MA, 2002; 88–91. ISBN 0201729156.
53. van Genuchten M, Hatton L. Quantifying software's impact. *Computer* 2013; **46**(10):66–72, doi:10.1109/MC.2013.7.
54. Tukey J. *Exploratory Data Analysis*. 1st edn., Addison Wesley, 1977. ISBN 02-010-7616-0.
55. Huber PJ, Ronchetti EM. *Robust Statistics*. 2nd edn., Wiley: Hoboken, NJ, 2009. ISBN: 978-0-470-12990-6.
56. Bellio R, Ventura L. An introduction to Robust Estimation with R functions. Current January 2017. Archived at WebCite as `http://www.webcitation.org/6nbWX34eu` October 2005. URL `https://www.researchgate.net/publication/228906268_An_introduction_to_robust_estimation_with_R_functions`, current July 2013.
57. RAE. The challenge of complex IT projects 2004. Royal Academy of Engineering report, London, ISBN 1-903496-15-2.
58. Economides N, Katsamakas E. Two-sided competition of proprietary vs. open source technology platforms and the implications for the software industry. *Management Science* 2006; **52**(7):1057–1071, doi:10.1287/mnsc.1060.0549.
59. MacCormack A, Rusnak J, Baldwin CY. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science* 2006; **52**(7):1015–1030, doi:10.1287/mnsc.1060.0552.
60. Boehm BW. *Software Engineering Economics*. Prentice Hall: Englewood Cliffs, NJ, 1981.
61. Maxwell K, Van Wassenhove L, Dutta S. Performance evaluation of general and company specific models in software development effort estimation. *Management Science* 1999; **45**(6):787–803, doi:10.1287/mnsc.45.6.787.
62. Schaller RR. Moore's law: past, present and future. *IEEE Spectrum* 1997; **34**(6):52–59.
63. Allan A, Edenfeld D, Joyner Jr WH, Kahng AB, Rodgers M, Zorian Y. 2001 technology roadmap for semiconductors. *Computer* 2002; **35**(1):42–53.
64. Mockus A, Weiss DM, Zhang P. Understanding and predicting effort in software projects. *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, IEEE Computer Society: Washington, DC, USA, 2003; 274–284.
65. Brooks FP. *The Mythical Man Month*. Addison-Wesley: Reading, MA, 1975.