

What Product Software Costs to Build and Own

Software Development and Maintenance Effort/Cost Models

Wayne Lobb, Engineering Director, David Warburton, Director of Systems Engineering and Tim Bosch, Vice President, Architecture & Consulting

Abstract

In Foliage's extensive experience developing products on contract, we have found that, on average over time, professional software R&D staffs create or modify about 700 lines of software logic per person per month, and own about 50,000 lines of software source code per person. The cost to build and own one line of product-software source for its whole lifetime ranges between \$10 to \$50 or more. These numbers depend on when and how lines and staff are counted. They also depend on numerous other factors, such as whether the software is safety-critical, or performance-critical, or was developed across multiple development sites. This paper describes how Foliage measures software size and categorizes staff for comparison purposes. The discussion concludes with recommended actions if your numbers appear to be out of line with the rest of your industry.

Is the inventory of product software in your company of a reasonable size, given your particular products and history? Is what you pay for your software in line with what others in your industry pay? Is your software R&D effort understaffed, overstaffed, or staffed in line with other similar companies that ship similar products? Judging where your inventory and costs rank depends on how staff is accounted for and on how software is measured and counted.

Foliage's counting and measuring rules are based on 20 years of experience designing and building commercial product software on contract. We have measured, assessed, and made comparative evaluations for hundreds of commercial software inventories. We build and evaluate many types of software, ranging from embedded to enterprise, from quick prototypes to large transactional systems, and from industrial control systems to safety-critical medical and aerospace systems, both pure-software products and software portions of larger electromechanical systems. We work with companies of all sizes and ages.

Foliage's key rule for measuring R&D staff size is to count all roles that "put hands on delivered software" including architects, developers, testers, build-deployment staff, documenters, bug-fixers, and multiple levels of management of all these.

Our key rule for counting lines of software source code is to include only essential software-product *logic* that your R&D staff members have built – or that you have outsourced to build – and not data or static configuration information that is embedded in software, and not stable third-party software for which you have the source, and so forth. The rationale for our approach is that custom logic consumes by far the largest portion of software effort from initial concept through end-of-life maintenance.

In sum, Foliage has found that, on average over time and across industries, a professional software R&D staff – including R&D management – generates about **700 lines per month per person of new or changed custom software logic**. This number applies, again on average, across the entire useful lifetime of a body of software. At \$150k per year on average for a fully burdened software staff person, a

line of software logic thus costs about \$18 in labor to build and own across its entire useful life. Counts and costs do vary: regulated software development in a large medical or aerospace company with a complex product line and multiple development sites worldwide could cost \$50 or more per line.

Also on average across industries and products, for a product that has shipped to customers and is being actively enhanced and supported, **each full-time development staff person accounts for about 50,000 lines of software source code**. That is, if your active product has 500,000 lines of source logic by Foliage's rules, then your company should have about 10 software R&D staff including management. There is considerable variation in this number too: for example, it tends to be significantly lower per person for safety-critical software or unusually complex software, and much higher for a product that is either under initial construction or nearing the end of its life.

And finally, almost always, **65-75 % of the total effort and labor costs come after first shipment** to paying customers. In other words, most of labor costs for software come from enhancements and fixes over long stretches of time, not in its initial construction. This is well known across industries but it is not always accounted for during planning.

There are many reasons why a company's software-inventory metrics *over time* might be very different from these numbers. Counting methods in your R&D might differ from Foliage's. Other common reasons are poor or obsolete architecture, unusually complex or fast-evolving products, or basic misunderstanding of the true costs of building and owning software. If your company's numbers are very different, then you should find out why.

Getting a Firm Handle on Your Software

In many companies that Foliage works with, true costs for developing and maintaining software often are not fully understood. Some symptoms of this misunderstanding within a company are uncorrected chronic cost and schedule overruns in development projects, as well as persistent quality or performance shortfalls in products.

Labor costs for developing the software in electro-mechanical products now often exceed labor costs for developing the hardware – which might be news. This change has come about during the careers of senior technical staff and executives. Many CTOs, CFOs, COOs and CEOs have had little early-career experience with software. They might not yet fully understand how product software development works, and how much the software shipped in products should cost to build and maintain. Software's invisibility contributes to misunderstandings. Some symptoms of misunderstanding within a company are uncorrected chronic cost and schedule overruns in software development projects, and persistent quality or performance shortfalls in software aspects of shipped products.

By way of example, a manufacturer of capital equipment for hospitals grew to a Fortune 1000 company by designing, manufacturing and selling common fixtures found in every hospital. These products were complex mechanical systems, and over the years the company became highly skilled and efficient in design, manufacturing and field service for such products. In the 1990s the company advanced from electromechanical controls to microprocessor-based controls, to gain flexibility, cost savings, and advanced features, and it added some firmware engineers to R&D.

Market drivers in the 2000s called for adding patient monitoring subsystems to the product line. This led to more software content and to a whole new group of software developers with new skills that went beyond firmware. The company expanded further into device connectivity solutions and equipment tracking solutions, which brought in yet more software content and yet other software skills. The company's product lines now have enterprise software elements, including relational databases and complex communications protocols that must provide totally reliable and secure information transactions – technologies that are far distant from electromechanical assemblies and devices.

Today, this company has more software engineers, and more software per capita across the entire company, than many purely-software product companies. But the company still does not apply deep insight and skill to managing product *software* in the way that it has historically managed product *hardware* in the form of electromechanical devices. There are still few or no equivalents on the software side for quality control, unit cost control, and inventory management.

For this company, software development remains essentially a black box.

Such a situation incurs unnecessary financial risks and competitive risks. But it is curable.

The cure starts with measuring and assessing the company's software inventory and developing effort/cost models. There is no one-size-fits-all model. It is essential that the inventory and the models represent *this* company's product profile, within the context of *its* particular sector, industry, and product line.

How Big Is Your Software Inventory?

The first and most fundamental thing that a product company should know is, just how much software do we own? Taking a useful and accurate inventory may seem like a straightforward task, but it is more complex than it appears. Here's why.

The “size” of software has to be viewed from the standpoints of both quantity and complexity. Complexity comes from the intricacy and criticality of the software logic needed to meet the product's requirements. The software logic within a quick prototype of a small commercial device can be simple and cost relatively little time and money to create and to test. By contrast, and in very different contexts, software that exercises 100% reliable decision support, or has extremely tight repeatability requirements, is almost always complex, difficult and expensive to build, though not always large in quantity. In regulated contexts, every possible contingency must be identified and dealt with, mistakes are unacceptable, and testing the software consumes a large fraction of the whole development budget.

Not all software should be counted in the inventory, and pure size – meaning lines of software source code – is by no means the whole story, as is discussed later in this paper.

The primary general rule Foliage uses when inventorying software, applying our own proprietary source-code-counting tool, is this:

- **Inventory Primary Rule:** In order to be counted in the software inventory, source code should have real logic in it that your software staff (or contracted software staff) have created and/or modified themselves, by hand, and will go on modifying and supporting as an essential ingredient in your shipped products.

By logic we mean conditional execution, e.g. “if XYZ condition is true then do A, otherwise do B”, or “repeat this set of commands for N number of times”. Source code that does not have logic in it is usually just data. Creating and maintaining data can be expensive. For example, it can be difficult and time-consuming to create known fixed inputs used to test a life-critical software function. But this work is of a different type than logic and has different, and much simpler, cost dynamics: once the data is ready it typically does not change significantly over time.

The Primary Rule’s qualifier of essential ingredient for countable code involves judgment. This is an area in which Foliage adds differentiable value to inventories: we have applied judgment across decades of development projects and have refined our methods continuously along the way, using feedback from real situations to tune the counting approach. There is no automated way to apply judgment – each case can be different from every other one. Listed below are some (but nowhere near all) of the secondary counting rules that Foliage applies:

- Third party code: Do not count third party source code used in your products unless your developers are modifying it on a regular basis. It’s not uncommon to use free open-source utility software within a product, modifying the utility code slightly to integrate it; such source code should *not* be counted in an inventory.
- Test code: Do not count test code unless a) it was developed specifically for this product by your developers, and b) it must be debugged and maintained significantly over time.
- Configuration files: Do not count configuration files that have no embedded logic; for example, do not count typical property=value (.ini) files or .xml data files.
- Data files: Do not count source files (C, C++, C#, Java etc) that are used only to insert initial or unchanging data into a program or database. More generally, as is expressed above as the primary rule, do not count files that have no actual logic in them other than assignment statements and/or conditional compilation flags.
- XML files: Do not count XML unless it has actual logic embedded in it, and if it does have embedded logic then count only the logic, not the other information. XML is mostly used for configuration and test data.
- XML transformations: Do count .xsl/xlst and other similar transformational logic that takes insight to develop, debug and maintain.
- Automatically-generated code: Do not count automatically-generated code as a rule, but certainly do count the fundamental logic that generates it. This can be complex given that modern graphical-user-interface (GUI) building tools, such as Microsoft Visual Studio, tend to mingle auto-generated configuration code with your manually-written code; in such contexts it might be best to apply a constant discount fraction to the sizes of those particular files.
- Unused code: Do not count duplicated or obsolete (dead) code. Note that it can be very difficult to detect unused code when much conditional compilation is involved.

You can begin a software inventory by having your software engineering staff measure your software by using free tools such as CLOC from SourceForge, or by using a commercial development tool such as Microsoft Visual Studio (MSVS), or Coverity’s powerful static analysis suite. The result will be an uncalibrated approximation of the count that a detailed approach like Foliage’s would generate.

For example, MSVS and Coverity will be highly accurate in their counts of actual used code in the shipped product. But these tools will often *not* include test code or other ancillary code that should be counted, and they usually *will* include large data-defining source files that Foliage normally rules out based on our experience.

The goal, as expressed above, is to account for all of the *logic*, and *only* the logic, that your software organization has itself built and will actively care for indefinitely. A quick count using off-the-shelf tools can misrepresent the total by as much as 50% in either direction, either too low or too high. Management needs the inventory size to be accurate in order to plan properly. Engineering needs the size to be accurate in order for their estimates and evaluations within the company to reflect the reality of the work they have done and have yet to do.

How Much Has It Cost to Build and Own Your Software?

Once the inventory count of lines of source code is in hand, the next step is to estimate how much your company has paid for the software it owns, both in terms of construction (or purchase) costs as well as follow-on costs to maintain, enhance and support the software.

While an exact accounting of software cost can be difficult and time consuming, Foliage has settled on a simple and practical model that considers, as a baseline proxy for total cost, only the *direct labor effort* inside the software R&D organization. We include the following roles in the calculation of effort:

- Software architects, designers, programmers, testers and documenters;
- Staff who operate source control systems and build/test/package systems – but *not* IT staff, and *not* staff who operate issue tracking systems unless the systems are completely owned and managed by core R&D;
- Software R&D maintenance staff who fix bugs and do minor enhancements, and *not* staff who support customers in the field, i.e. not application engineers, even though those engineers might create large amounts of software in their own right;
- Management of all the above groups, up to and including the highest software technical management positions within engineering or R&D.

Direct labor effort is a good proxy for total cost because expenses for development equipment and related tools have steadily gone down over the past two decades. A high performance computer now costs only a small fraction of the typical annual fully-burdened cost for a software developer, perhaps \$4-5k for a high-end machine compared to \$130-200k annually on average for a high-end developer. There are very capable free tools available for developing code, debugging code, doing code analyses, managing source control, doing version control, tracking issues, enabling database access, and so forth. Your situation may require expensive commercial tools, but if so then your software organization is probably quite large, numbering 100 or possibly many more, and the tool expenses will be amortized over a wider base.

Your staff members can most likely estimate, with reasonable ease and accuracy, how many developer-years have gone into creating the software in your inventory, accounting for the set of roles listed above. The investment to date in your inventory can then be calculated by applying average fully-burdened annual costs. Many executives are surprised by how large these numbers turn out to be.

How Do Your Inventory Size and Efforts/Costs Compare?

Foliage has found that, *on average and over time*, software staff members who are full-time employees can produce and support around 700 new or changed lines of software *logic* per person-month, or equivalently some 8,000 lines of software logic per person-year.

These productivity numbers may seem very low. They have to be interpreted, and used with care, in light of the following assumptions and characterizations:

- **Physical versus logical lines:** Foliage's proprietary tool for measuring source code size counts only source lines and comment lines that have substantial content, i.e. not blank lines or trivial lines that have fewer than three characters in them. This results in line-count totals that are smaller than raw physical line counts in files, and larger than so-called logical line counts that many other source-code measurement tools generate.
- **File types and staff roles:** The software files included in the line counts are as listed earlier, and the software staff included in the effort totals are as listed earlier as well.
- **Retrospective and not as-you-go:** These numbers are *retrospective*. They include extended maintenance and support of software that may have been around for many years, even decades. As is discussed below in this paper, *software maintenance and support cost far more over time than original software development* – this may be highly counterintuitive.
- **Variation due to larger product context:** There is much variation from case to case. Regulated software development is more labor-intensive than the average, and has correspondingly lower productivity numbers. Specialized organizations like Foliage are able to maintain significantly higher productivity numbers than these, mainly because we work only on developing new software and restructuring bodies of software, and not on extended maintenance and support.

Foliage has also found that, on average over the lifetime of a body of software, that full-time software R&D employees each account for about 50,000 lines of software logic source code. Before first shipment to customers the total per person may be significantly higher than this, before maintenance begins and support staff become involved. Late in the lifecycle the number of lines per person will increase again, when little if any new development is happening and nearly all the work is in minor fixes.

Here's an example of software size and staff productivity. Much of Foliage's software development work in the past 20 years has been in control systems for large capital equipment products. Such a system might have 800k lines of code (by Foliage's counter) and be 7 years old. At \$150k per R&D staff member per year, and at 600 lines per staff-month, this comes out to a total *retrospective* investment to date of 111 staff-years, \$16.7 million, \$21 per line of code, an annual average of 16 employees in the software group including management, and about 50k lines of existing code per software staff person again including management. Equipment control systems are harder to develop than average software, and productivity at 600 lines per month is expected to be lower than the whole industry average. 50k lines per software staff is a typical number over time. If your software is similar to this but your lines-per-staff or dollars-per-line are radically different, then chances are that you're either wasting money on software or underfunding software. In the latter case you will

almost certainly be experiencing ongoing problems with unpredictable delivery dates and low quality.

The software industry has a long history of *predictive* parameterized cost models that account for the inherent difficulty of the problem to be solved. These models apply contextual variables like regulatory constraints that affect costs, plus the skills and experience of the group that will build (or did build) the software. See for example the constructive cost models (COCOMO) developed and refined at USC over the past thirty years.

In Foliage's experience, there are at least nine significant differentiators for productivity, effort and cost:

1. **Safety-critical software:** Regulated development for safety-critical usually runs closer to 400 lines per software staff-month. A highly skilled and experienced software group in a regulated environment can produce much more than this, but only over time and with very strong management. In the best case, regulated development should require no more than 10% extra effort and cost beyond the average numbers.
2. **Organization size:** Productivity in large organizations – 100 software staff and up – tends to be significantly lower than in small organizations. Communications overhead grows rapidly with increasing size unless managed extremely well. Staff members in large organizations tend to multi-task more with multiple responsibilities, and to suffer more ramp-up/ramp-down loss of productive time. Large software bases are always mature ones and more effort overall will be going into maintenance as opposed to production of new code. Foliage has observed on the order of 25% lower productivity in large organizations.
3. **Distributed development:** Software developed by teams that are not physically proximate costs more to build than teams that are fully co-located. This is because, as in large organizations, the communications overhead is higher across geographical, time-zone and cultural distances. On the other hand, there may be very good reasons to distribute the development among sites, for example to “follow the sun” around the world in a very dynamic development scenario, or to have certain kinds of development close to end customers for faster feedback and a better cultural fit. Cost savings are rarely as great as one might expect for offshore development: lower hourly rates are counterbalanced by lower skill levels, more staff turnover volatility, the need for more extensive and rigorous requirements to be provided, and, again, the various overhead costs of distance.
4. **Organization skill and experience:** Excellent developers build less code to accomplish a task than less-capable developers. They also build the code much faster, and the code they produce has far fewer problems. The difference in problem-free output between the most capable developers and the least capable developers can be as high as 10:1.

5. **Staff focus:** Productivity tends to be significantly higher than average in a startup or at a company like Foliage that only builds software and systems products on contract. Engineers in these situations rarely have to multi-task and can sustain very high levels of productivity when building new systems. This places the tightly focused design teams at contract companies like Foliage at a cost advantage to internal teams when the development cost is measured on a cost per line basis. Focused teams have two other major advantages: the speed with which a new product can be brought to market, and the lower numbers of defects and issues in software developed by a highly experienced, fully focused team.
6. **Complexity:** The more complex the problem is that's being solved, the more complex the software generally has to be. This leads to lower *apparent* productivity in terms of lines of code. Software complexity is most commonly measured by counting the number of different paths through a body of software logic; the total number is known as the McCabe metric for that body of code. The McCabe metric can be misleading: for example, a large case statement may be totally simplistic conceptually but get counted as highly complex. It is essential that your company understand the true inherent complexity of the problems you are solving, and the truly necessary complexity of the software you are producing, before evaluating productivity on a lines-of-code basis. One way to do this is to have outside experts help you gauge your specific challenges as compared with other companies in your industry.
7. **Development for re-use:** Software that is built for a single product or for a single purpose is significantly less complex than software that is intended to be re-used in multiple contexts, e.g. in several products in a product line. From a distance, it can seem that re-use should be simple to achieve given the similarity of functions between different products, but the similarity is often only superficial, and design for re-use will be much more costly than anticipated. In the best cases, development for re-use typically incurs a 100% premium in effort and cost, which means that in practice it's usually better to build twice for two uses and seek re-use only when there are three or more distinct but not fundamentally different usages.
8. **Robustness of the architecture:** A hastily built body of software that has not been carefully designed to be optimal with respect to its requirements will likely cost far more overall than it needed to during its lifetime. This is because a poorly-fitted system, whatever its context, will have more friction and more problems than a well-fitted one. Costs can skyrocket where there's a bad fit. An excellent original architecture can degrade over time if developers do not follow its design principles and instead take shortcuts when enhancing or fixing the code.
9. **Evolution versus new development:** Productivity rates based on line counts are much higher than average for software development that refurbishes old or existing software. These evolutionary projects typically migrate to new technologies or infrastructures, or expand internal structures to enable advanced features or higher performance. In these cases, the monthly effort-line numbers, in Foliage's experience, can be as high as 2,000 lines per staff-month, and in some outlier cases as high as 10,000 lines per staff-month

Where Does the Money Go?

When you examine your software inventory and how much you've paid for it, you will probably be interested in where the effort and the money have gone over the years.

As Figure 1 illustrates, the fundamental fact about commercial software is this:

*On average, about $\frac{3}{4}$ of the R&D labor costs for a body of software are incurred **after** the software has first been shipped to paying customers as a product or part of a product: most of the labor and cost of software is in maintenance and support.*

See for example <http://www.sei.cmu.edu/reports/04sr003.pdf>, which is dated but still valid.

Software is almost never fully complete at the time that it first ships to a paying customer. The longer a body of software remains commercially viable, the higher the fraction of total effort that will go into enhancing it, fixing it and supporting it during its lifetime. Changes in third-party code may force changes within your software even if your software is stable in all other respects. New interfaces have to be created to meet evolving requirements. And the more corner cutting there was during initial development – for example skimping on feasibility prototypes or on architectural analysis and design – the higher the costs will be during enhancements and maintenance, further increasing the fraction of post-shipment effort and costs.

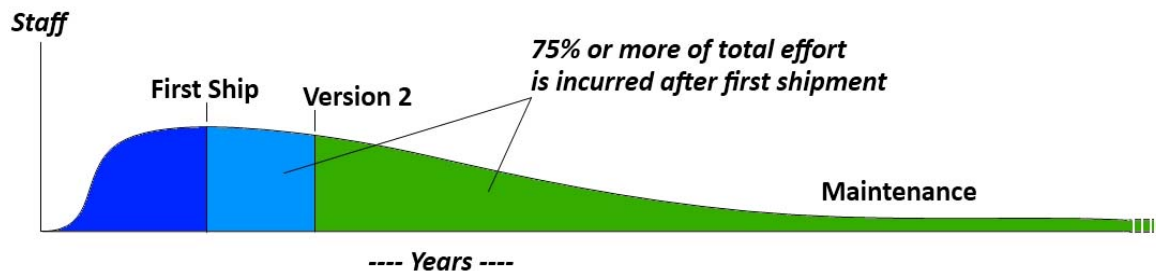


Figure 1. Staff Effort on Software through the Lifecycle

In the previous equipment control system example, 800k lines of code were developed by a staff that averaged 16 permanent employees (including management) across 7 years of development and maintenance. Chances are that the software was shipped to its first paying customer at about the end of three years of development effort, and that the original development team was more like 20 or 25 people. For about two years after first shipment, almost the whole team continued to work full-time on the software as it matured in features and stability, and as more and more customers began using it. For the most recent two years, the team working on this software likely scaled back by perhaps half, with the remainder moving over to new development. It's likely that this control system software will continue to be used in the field for another 8 to 10 years, or even much longer, and that over time the number of staff members working on it will continue to dwindle.

When the numbers are run, it will become apparent that most of the software effort, and most of the total cost of software, came *after* the first shipment. This is true for basically all product software. This fact should be a bedrock consideration when understanding software costs, especially when estimating costs and schedules your

next generation of software development. It is true across all kinds of applications and all industries, and true of all development methods including agile methods.

Know What Your Size and Effort Numbers Mean

As described above, your numbers may vary for many reasons. If your R&D group has followed the counting rules that Foliage uses (not all of our rules are described in this paper) and has come up with what seem to be off-scale numbers, then the first thing to look into is whether the counts are accurate. After that, investigate whether your software architecture was carefully designed and has been adhered to since the product first shipped. Then consider whether you are understaffed or overstaffed based on software size and complexity. Foliage can help.

The best predictor by far of what new software will cost is your own company history with software efforts and costs, given the specific characteristics of your situation. Industry models are useful and informative, but unless you examine and analyze your own historical data and adjust your plans and estimates accordingly, your software development will continue to be a black box, and your software costs and schedules will likely be out of control.

About Foliage

Foliage is a leading provider of custom product development and product strategy consulting to the Aerospace, Medical Devices and Life Sciences, Manufacturing Systems, and Semiconductor / Solar industries. With experience earned by successfully delivering over 300 products to market, Foliage delivers a potent time-to-market advantage and maximizes this by optimizing quality, reducing costs, and minimizing risks. A privately held company, Foliage was founded in 1991 and is headquartered in Burlington, Massachusetts. For more information, visit: www.foliage.com.

168 Middlesex Turnpike, Burlington, MA 01803 781.993.5500
Copyright 2011 Foliage, Inc.

