# Using Artificial Intelligence (AI)
# For Large Software Engineering Projects – Part 3

Capers Jones

**Foreword:** The following pages constitute Part 3 of *Using Artificial Intelligence for Large Software Engineering Projects* by Capers Jones, released for distribution to the International Function Point Users Group (IFPUG) in January of 2025. Topics in this extract include 26 Improvements for the Next Five Years.

**Note**: Many of the illustrations in the original document were produced using artificial intelligence. As of February, 2025 AI-generated illustrations cannot be copyrighted. The remaining content of this document, while shared, is copyrighted by Capers Jones, 2025.

**Acknowledgment**: The IFPUG community expresses its appreciation to Mr. Jones for his lifelong pursuit of metrics-based state of and recommended improvements for software development practices globally.

Joe Schofield

**Short-Term Improvements for the Next Five Years**

Although major improvements based on high volumes of standard reusable software components may not occur soon, there are a number of short-term improvements that could be achieved over the next five years:

The author suggests that every major software-producing company and government agency have their own set of five-year targets, using the current list as a starting point.

1. **Raise defect removal efficiency (DRE) from < 92.50% to > 99.50%.** This is the most important goal for the industry. It cannot be achieved by testing alone but requires pre-test inspections and static analysis. Automated proofs and automated analysis of code are useful. Certified test personnel and mathematical test case design using techniques such as design of experiments are useful. The DRE metric was developed by IBM circa 1970 to prove the value of inspections. It is paired with the "defect potential" metric discussed in the next paragraph. DRE is measured by comparing all bugs found during development to those reported in the first 90 days by customers. The current U.S. average is about 90%. Agile is about 92%. Quality strong methods such as RUP and TSP usually top 96% in DRE. Only a few top companies using a full suite of defect prevention, pre-test defect removal, and formal testing with mathematically designed test cases and certified test personnel can top 99% in DRE. The upper limit of DRE circa 2015 is about 99.6%. DRE of 100% is theoretically possible but has not been encountered on more than about 1 project out of 10,000.

2. **Lower software defect potentials from > 4.25 per function point to < 2.0 per function point.** The phrase "defect potentials" was coined in IBM circa 1970. Defect potentials are the sum of bugs found in all deliverables: 1) requirements, 2) architecture, 3) design, 4) code, 5) user documents, and 6) bad fixes. Requirements and design bugs often outnumber code bugs. Today defect potentials can top 6.0 per function point for large systems in the 10,000 function point size range. Achieving this goal requires effective defect prevention such as joint application design (JAD), quality function deployment (QFD), requirements modeling, certified reusable components, and others. It also requires a complete software quality measurement program. Achieving this goal also requires better training in common sources of defects found in requirements, design, and source code. The most effective way of lowering defect potentials is to switch from custom designs and manual coding, which are intrinsically error prone. Construction for certified reusable components can cause a very significant reduction in software defect potentials.

3. **Lower cost of quality (COQ) from > 45.0% of development to < 15.0% of development.** Finding and fixing bugs has been the most expensive task in software for more than 50 years. A synergistic combination of defect prevention and pre-test inspections and static analysis are needed to achieve this goal. The probable sequence would be to raise defect removal efficiency from today's average of below 90% up to 99%. At the same time defect potentials can be brought down from today's averages of more than 4.0 per function point to less than 2.0 per function point. This combination

will have a strong synergistic impact on maintenance and support costs. Incidentally lowering cost of quality will also lower technical debt. But as of 2014 technical debt is not a standard metric and varies so widely it is hard to quantify.

4. **Reduce average cyclomatic complexity from > 25.0 to < 10.0.** Achieving this goal requires careful analysis of software structures, and of course it also requires measuring cyclomatic complexity for all modules. Since cyclomatic tools are common and some are open source, every application should use them without exception. In the age of artificial intelligence another way of lowering complexity will be 3D diagrams of system structures using 3D screens:

5. **Raise test coverage from < 75.0% to > 98.5% for risks, paths, and requirements.** Achieving this goal requires using mathematical design methods for test case creation such as using design of experiments or cause-effect graphing. It also requires measurement of test coverage. It also requires predictive tools that can predict numbers of test cases based on function points, code volumes, and cyclomatic complexity. The author's Software Risk Master (SRM) tool predicts test cases for 18 kinds of testing and therefore can also predict probable test coverage.

6. **Eliminate error-prone modules (EPM) in large systems.** Bugs are not randomly distributed. Achieving this goal requires careful measurements of code defects during development and after release with tools that can trace bugs to specific modules. Some companies such as IBM have been doing this for many years. Error-prone modules (EPM) are usually less than 5% of total modules but receive more than 50% of total

bugs. Prevention is the best solution. Existing error-prone modules in legacy applications may require surgical removal and replacement. However static analysis should be used on all identified EPM. In one study a major application had 425 modules. Of these 57% of all bugs were found in only 31 modules built by one department. Over 300 modules were zero-defect modules. EPM are easy to prevent but difficult to repair once they are created. Usually, surgical removal is needed. EPM are the most expensive artifacts in the history of software. EPM is somewhat like the medical condition of smallpox; i.e. it can be completely eliminated with "vaccination" and effective control techniques. Error-prone modules often top 3.0 defects per function and remove less than 80% prior to release. They also tend top 50 in terms of cyclomatic complexity. Higher defect removal via testing is difficult due to the high cyclomatic complexity levels.

7. **Eliminate security flaws in all software applications.** As cyber-crime becomes more common the need for better security is more urgent. Achieving this goal requires use of security inspections, security testing, and automated tools that seek out security flaws. For major systems containing valuable financial or confidential data, ethical hackers may also be needed.

8. **Reduce the odds of cyber-attacks from > 10.0% to < 0.1%.** Achieving this goal requires a synergistic combination of better firewalls, continuous anti-virus checking with constant updates to viral signatures; and also increasing the immunity of software itself by means of changes to basic architecture and permission strategies. It may also be necessary to rethink hardware and software architectures to raise the immunity levels of both. Moving from custom manual development to construction from certified and hardened reusable components is on the critical path for cyber-attack reduction.

9. **Reduce bad-fix injections from > 7.0% to < 1.0%.** Not many people know that about 7% of attempts to fix software bugs contain new bugs in the fixes themselves commonly called "bad fixes." When cyclomatic complexity tops 50 the bad-fix injection rate can soar to 25% or more. Reducing bad-fix injection requires measuring and controlling cyclomatic complexity, using static analysis for all bug fixes, testing all bug fixes, and inspections of all significant fixes prior to integration.

10. **Reduce requirements creep from > 1.5% per calendar month to < 0.25% per calendar month.** Requirements creep has been an endemic problem of the software industry for more than 50 years. While prototypes, agile embedded users, and joint application design (JAD) are useful, it is technically possible to also use automated requirements models to improve requirements completeness. The best method would be to use pattern matching to identify the features of applications similar to the one being developed. A precursor technology would be a useful taxonomy of software application features, which does not actually exist in 2015 but could be created with several months of concentrated study.

11. **Lower the risk of project failure or cancellation on large 10,000 function point projects from > 35.0% to < 5.0%.** Cancellation of large systems due to poor quality, poor change control, and cost overruns that turn ROI from positive to negative is an endemic problem of the software industry, and totally unnecessary. A synergistic combination of effective defect prevention and pre-test inspections and static analysis can come close to eliminating this far too common problem. Parametric estimation tools that can predict risks, costs, and schedules with greater accuracy that ineffective manual estimates are also recommended. AI-generated designs can help visualize areas that may need improvements:

12. **Reduce "bad fix" injections from around 5% to less than 1% for large systems.**

13. **Reduce the odds of schedule delays from > 50.0% to < 5.0%.** Since the main reasons for schedule delays are poor quality and excessive requirements creep, solving some of the earlier problems in this list will also solve the problem of schedule delays. Most projects seem on time until testing starts, when huge quantities of bugs begin to stretch out the test schedule to infinity. Defect prevention combined with pre-test static analysis can reduce or eliminate schedule delays. This is a treatable condition and it can be eliminated within five years.

14. **Reduce the odds of cost overruns from > 40.0% to < 3.0%.** Software cost overruns and software schedule delays have similar root causes; i.e. poor quality control and poor change control combined with excessive requirements creep. Better defect prevention combined with pre-test defect removal can help to cure both of these endemic software problems. Using accurate parametric estimation tools rather than optimistic manual estimates are also useful in lowering cost overruns.

15. **Reduce the odds of litigation on outsource contracts from > 5.0% to < 1.0%.** The author of this paper has been an expert witness in 12 breach of contract cases. All of these cases seem to have similar root causes which include poor quality control, poor change control, and very poor status tracking. A synergistic combination of early sizing and risk analysis prior to contract signing plus effective defect prevention and pre-test defect removal can lower the odds of software breach of contract litigation.

16. **Lower maintenance and warranty repair costs by > 75.0% compared to 2016 values.** Starting in about 2000 the number of U.S. maintenance programmers began to exceed the number of development programmers. IBM discovered that effective defect prevention and pre-test defect removal reduced delivered defects to such low levels that maintenance costs were reduced by at least 45% and sometimes as much as 75%. Effective software development and effective quality control has a larger impact on maintenance costs than on development. It is technically possible to lower software maintenance for new applications by over 60% compared to current averages. By analyzing legacy applications and removing error-prone modules plus some refactoring it is also possible to lower maintenance costs by legacy software by about 25%. Technical debt would be reduced as well, but technical debt is not a standard metric and

varies widely so it is hard to quantify. Static analysis tools should routinely be run against all active legacy applications.

17. **Improve the volume of certified reusable materials from < 15.0% to > 35.0%.** Custom designs and manual coding are intrinsically error-prone and inefficient no matter what methodology is used. The best way of converting software engineering from a craft to a modern profession would be to construct applications from libraries of certified reusable material; i.e. reusable requirements, design, code, and test materials. Certification to near zero-defect levels is a precursor, so effective quality control is on the critical path to increasing the volumes of certified reusable materials. All candidate reusable materials should be reviewed, and code segments should also be inspected and have static analysis runs. Also, reusable code should be accompanied by reusable test materials, and supporting information such as cyclomatic complexity and user information.

18. **Improve average development productivity from < 8.0 function points per month to >16.0 function points per month.** Productivity rates vary based on application size, complexity, team experience, methodologies, and several other factors. However, when all projects are viewed in aggregate average productivity is below 8.0 function points per staff month. Doubling this rate needs a combination of better quality control and much higher volumes of certified reusable materials; probably 50% or more.

19. **Improve work hours per function point from > 16.5 to < 8.25.** Goal 17 and this goal are essentially the same but use different metrics. However there is one important difference. Work hours per month will not be the same in every country. For example a project in the Netherlands with 116 work hours per month will have the same number of work hours as a project in China with 186 work hours per month. But the Chinese project will need fewer calendar months than the Dutch project due to the more intense work pattern. Note that AI can drop work hours per function point to below 1.0.

20. **Improve maximum productivity to > 100 function points per staff month for 1000 function points.** Today in early 2016 productivity rates for 1000 function points range from about 5 to 12 function points per staff month. It is intrinsically impossible to top 100 function points per staff month using custom designs and manual coding. Only construction from libraries of standard reusable component can make such high productivity rates possible. However, it would be possible to increase the volume of reusable materials. The precursor needs are a good taxonomy of software features, catalogs of reusable materials, and a certification process for adding new reusable materials. It is also necessary to have a recall method in case the reusable materials contain bugs or need changes. AI can generate order of magnitude improvements.

21. **Shorten average software development schedules by > 35.0% compared to 2024 averages.** The most common complaint of software clients and corporate executives at the CIO and CFO level is that big software projects take too long. Surprisingly it is not hard to make them shorter. A synergistic combination of better defect prevention, pre-test static analysis and inspections, and larger volumes of certified reusable materials can make significant reductions in schedule intervals. In today's world raising software application size in function points to the 0.4 power provides a useful approximation of schedule duration in calendar months. But current technologies are sufficient to lower the exponent to the 0.37 power. Raising 1000 function points to the 0.4 power indicates a schedule of 15.8 calendar months. Raising 1000 function points to the 0.37 power shows a schedule of only 12.9 calendar months. This shorter schedule is made possible by using effective defect prevention augmented by pre-test inspections and static analysis. Reusable software components could lower the exponent down to the 0.3 power or 7.9 calendar months. Schedule delays are rampant today, but they are treatable conditions that can be eliminated. AI can reduce schedules by over 80%/

22. **Raise maintenance assignment scopes from < 1,500 function points to > 5,000 function points.** The metric "maintenance assignment scope" refers to the number of function points that one maintenance programmer can keep up and running during a calendar year. This metric was developed by IBM in the 1970's. The current range is from < 300 function points for buggy and complex software to > 5,000 function points for modern software released with effective quality control. The current U.S. average is about 1,500 function points. This is a key metric for predicting maintenance staffing for both individual projects and also for corporate portfolios. Achieving this goal requires effective defect prevention, effective pre-test defect removal, and effective testing using modern mathematically based test case design methods. It also requires low levels of cyclomatic complexity. Static analysis should be run on all applications during development, and on all legacy applications as well.

23. **Replace today's static and rigid requirements, architecture, and design methods with a suite of animated design tools combined with pattern matching.** When they are operating, software applications are the fastest objects yet created by the human species. When being developed, software applications grow and change on a daily basis. Yet every single design method is static and consists either of text such as story points or very primitive and limited diagrams such as flowcharts or UML diagrams. The technology for created a new kind of animated graphical design method in full color and also three dimensions exists today in 2023. It is only necessary to develop the symbol set and begin to animate the design process.

24. **Develop an interactive learning tool for software engineering based on massively interactive game technology.** New concepts are occurring almost every day in software engineering. New programming languages are coming out on a weekly basis. Software lags medicine and law and other forms of engineering in having continuing education. But live instruction is costly and inconvenient. The need is for an

interactive learning tool with a built-in curriculum planning feature. It is technically possible to build such as tool today. By licensing a game engine it would be possible to build a simulated software university where avatars could both take classes and also interact with one another.

25. **Develop a suite of dynamic, animated project planning and estimating tools that will show growth of software applications.** Today the outputs of all software estimating tools are static tables augmented by a few graphs. But software applications grow during development at more than 1% per calendar month and they continue to grow after release at more than 8% per calendar year. It is obvious that software planning and estimating tools need dynamic modeling capabilities that can show the growth of features over time. They should also the arrival (and discovery) of bugs or defects entering from requirements, design, architecture, code and other defect sources. The ultimate goal, which is technically possible today, would be a graphical model that shows application growth from the first day of requirements through 25 years of usage.

26. **Introduce licensing and board certification for software engineers and specialists.** It is strongly recommended that every reader of this paper also read Paul Starr's book The Social Transformation of American Medicine. This book won a Pulitzer Prize in 1982. Starr's book shows how the American Medical Association (AMA) was able to improve academic training, reduce malpractice, and achieve a higher level of professionalism than other technical field. Medical licenses and board certification of specialists were a key factor in medical progress. It took over 75 years for medicine to reach the current professional status, but with Starr's book as a guide software could do the same within 10 years. This is outside the 5-year window of this article, but the process should start in 2025.

These short-term targets are within the reach of almost all major software corporations, if they want to improve.