*Jay Cowan*

*[Company Name] | [Company Address]*

*Measuring Software Engineering*

Measuring, Evaluating, Calculating & Considering Software Engineering Performance

# Introduction

Software engineering is a new, broad, and dynamic field. Unlike physics, biology, or chemistry, which can trace their origins to legends of history like Pythagoras or Hippocrates, software engineering as a field of practice is arguably around half a century old. As a result, the styles and application of the field is rather loose and has a diversity of approaches and theories. Some even question whether software engineering is a form of engineering at all, with notable figures in computer science like Edsger Dijkstra titling the field “How to program if you cannot” [1]. Despite this, in the practical world of software, software engineering has been embraced as a field of practice taken seriously by governments, international organizations, and the largest and most valuable companies in the world.

Many approaches and methodologies to organize and further the practice of software engineering has been developed to handle the ever-increasing scale and complexity of modern software projects. Agile, Waterfall, Rapid and DevOps have all been introduced as practices to best handle development, maintenance, and deployment of projects from small-scale applications to enterprise products. Around these methodologies, many tools have been developed to implement and improve these practices, notably the Atlassian suite and GitHub Enterprise. Many of these tools have been used to improve measurement and organization of software projects.

All of these tools, practices and methodologies seek to accomplish but a few simple goals, that being creating a measurable and systematic approach to software development, applying the precepts of engineering to software development. Maintenance, business, timelines and other concerns factor into the day-to-day practical considerations of the software engineering world.

# Measuring Engineering Activity

Of the various methodologies involved in the process of software development, all require some form of quantification to determine when a stage of the process is complete. Waterfall is only really complete on release of the project, with maintenance following suit. Rapid application development works in cycles of prototyping, demonstrating and defining, with completion at client approval and deployment. Agile cycles are considered complete at the end of their sprint finishing one feature or aspect to the project, leading to the next one. DevOps takes this another step by focusing on continuous integration and deployment, releasing small implementations often, and automating builds and releases as much as possible. When it comes down to the core of each of these strategies, it is clear that the main issue that they focus on is quantification.

When is a feature actually done? Is it when you have a working implementation that passes tests? Or is it only when the project is released in its entirety? And how can you know if someone is contributing their fair share? It’s obviously difficult to compare the contributions of a seasoned developer on the team to the new hire who has been on the team barely a week. Settling on simple solutions like commits or passing automated test suites can lead to a moral hazard where developers optimize their work to just look good, rather than actually getting the job done. Thus we must consider more holistic methods to measure engineering progress, and the extent to which the various methodologies appropriately quantify progress. Additionally, the class of project at issue will consequently become a major consideration to the software engineering process.

Some ways that we can think about the types of projects that affect how we approach the measurement of software engineering starts with timelines. A triple-A game will have very different measurement considerations than, for example, a CRM platform. Where your triple-A developer is looking towards the release date, and thus the final product at that date, the CRM developer is more concerned about the singular feature and will accept a slower speed of development if it creates a better end product. Naturally different development methodologies better fit each of these strategies, perhaps waterfall for the triple-A game studio and Agile or DevOps for the CRM development team, and thus a one-size-fits all approach is unwise for the issue of progress measurement.

Segmenting performance quantification and finding appropriate scale for tasks in a project can be an important factor for effectively measuring software engineering performance. Tasks with too wide of a scope lack direction, leading to overlapping roles and stymied progress. Too narrow of a scope can create a false sense of achievement, and waste time by bogging the team down in design considerations, without making considerable progress on the project as a whole. Finding appropriate scope requires project directors to set defined and achievable goals that fall within a scope.

For example, a team may be building a calendar as part of a larger system, and when determining the scope, may want to set the calendar as a whole, with all of the integrations and extra features as a single agile iteration. Another director may set the focus too small, suggesting that the first iteration focus on the event modal, then the next on the dropdown, then the next on a settings pane and so on and so on, wasting time on small features that can be enveloped in an achievable but focused goal. This team would be better off by focusing on getting a working calendar first, then focusing on the extra features at the next iteration or set of iterations, making progress where they can, and avoiding getting stymied or lost in too broad of a scope. This approach is a functional and more easily measurable form of segmentation appropriate for development teams rather than a single developer.

When it comes to measuring the progress of a single developer, focusing on the human issue is paramount. Any statistician worth their salt will tell you that a univariate analysis is always missing critical information. Likewise, focusing on singular or simple factors like the number of code commits, lines written or total characters written will be absent of any real meaning and can be effectively exploited by the people they are supposed to evaluate. People naturally want to look productive and accomplished, and of course will leverage simple factors that contributes to this, and if they are evaluated on, for example, the number of commits they push, then they will increase volume, without improving output or quality, as this is in their best interest. Thus, simple approaches like this are best avoided for critically evaluating output and performance and absent other factors are largely meaningless. Beyond this, there are many other reasons why simple analysis may fall short, as a developer may be exceptional at cleaning up an optimizing code, but below par in its initial creation. These human factors emphasize the need for holistic and flexible personal evaluation approaches to software engineering.

Where a business manager may seek to build a formulaic approach tied to revenue and sales, seasoned developers know that there are more considerations beyond simply the monetary value of what they are currently working on. Other firms may be focused on the number of lines of code a single developer produces, but again, this is a flawed conception, as a significantly better developer would likely produce a much more concise and maintainable codebase.