# Measuring Software Engineering Report

**Name** Nadia Abouelleil

**Student Number** 18340116

# Section 1: Measuring Engineering Activity

Modern software engineering has evolved to included software metrics, an increasingly important area in the engineering sphere. Contractual responsibilities to customers that include software and/or quality metrics reporting are becoming necessary for modern businesses. Businesses utilize metrics to understand, manage, control, and predict software projects, processes, and products. Since we're assessing the software engineering process directly, I'll take a more sophisticated approach to this subject. Any person involved in the selection, design, implementation, collection, or use of a metric must understand its meaning and purpose. To obtain useful data, we must examine factors such as project cost and effort forecasting, root cause analysis, a specific test coverage indicator, and computer performance modelling.

Identifying entities is the first step. To measure something, we must first identify it. Assessments are often used in production to estimate costs, identify issues, assess quality, and track progress. A hypothetical example of this for software engineering: choosing an API endpoint as our entity. After selecting an entity, we must assign it an attribute that we want to describe. In this case, our entity's speed in processing queries and sending its results may be the two properties. Finally, we need a well-defined and broadly approved mapping mechanism. No one can claim that the processing speed of the API is 15 or that the API's response time is 10 unless they understand the terms. For successful measurement, we must specify these three steps correctly. We risk endangering the entire process of evaluating engineering processes if we don't calibrate these 3 components of measurement correctly.

The input-process-output paradigm uses the measurement model as the key building block to software entities. Software entities of the input type includes all resources used for development, production, and software research. Software entities of the processes type include software-related actions and occurrences that are often time-related. Process entities include defined actions such as building a software system from specifications through customer delivery, checking code, and the 1st 6 months after delivery of the product. We may want to evaluate a source code's testability, usability, reliability, or maintainability. The software process's products are software entities. Software output entities include test documents (i.e., specs, plans, reports, cases, scripts), progress reports, budgets, project plans, and problem reports.

Now that we know what makes up the software engineering process, we can learn about the many methods for measuring it. It's all about measuring function analysis. The measuring function defines the metric's calculating mechanism. Some metrics are developed mathematically from simple measurements (e.g., equations or algorithms). The inspection's preparation rate, calculated by multiplying the number of lines inspected by the

number of preparation hours, is a derived metric. Many measurement models are simple. Using modelling offers both benefits and drawbacks. We must be pragmatic while creating a model for measuring. Trying to include all elements that influence a property or characterize an entity quickly renders our model useless. Pragmatism means not creating the most thorough model possible. It means prioritizing the most vital elements.

## Section 2: Platforms for collecting and analysing data sets

Software metrics can be classified as static or dynamic. One says to collect data on everything before examining it for meaning or correlation. This strategy collects massive volumes of data and performs a stricter trend analysis. The second line of reasoning is the shotgun approach to measurements. This mainly involved acquiring and reporting on current "hot" metrics or creating metrics using data generated by software development. This "hot" method analyses current events to determine the current software engineering stage. Dynamic assessment is a modern evaluative technique. Dynamic assessment says, “We need to measure what's going on right now, if what's going on now changes, our assessment changes”. It's a more flexible strategy. Goals and standards should be set using this process. This week or two weeks, we'd figure out what needed to be done, and then how to gauge our effectiveness. If we go this way, we'll have to create new goals and organize new metrics. A static strategy can help us achieve our goals faster. This strategy's linear evolution makes it easier to analyse and evaluate. Because this strategy prevents change, we may utilize it to set up permanent testing procedures, collect data, and establish standards. This technique is easier, less resource-intensive, and more reliable because we can establish a database of analytics.

Both approaches have major problems. The issue with the static method is that there are just too many metrics when all possible software entities and characteristics are considered. It's easy for a company to become overwhelmed by the effort of measuring, tracking, and forecasting every aspect of our operation. In one of my favourite lines, "spending all of our time reporting on the nothing we are doing because we are spending all of our time reporting." For example, Watts Humphrey points out that there are so many possible measurements in a complex software process that picking one at random is unlikely to generate anything useful. To improve our approach, we should maintain our methods for monitoring and evaluating software engineering flexible.

Modern Assessment is based on a fundamental shift in software metrics philosophy. Software metrics solutions are rapidly being created to provide exact data for software project management and process improvement. Metrics are chosen depending on organizational, project, and task goals. These metrics measure our success in reaching our goals. This strategy just measures what we need to know. Instead of asking "What should I measure?" this strategy asks, "Why am I measuring?" or "What business needs does the organization want its measurement project to address?"

***Case Study:***

Contrast the two most widely used software engineering methodologies for monitoring and analysing the standard of the software engineering processes they cause. This will impact how the developer is evaluated.

**Agile Assessment** - The needs of the client and the development team change over time. This strategy aligns development and testing with client needs. Among the benefits of Agile Testing are: Simple structure necessitates minimum forethought. There's not much info. Testers work with developers on the project. Smaller tasks fit best. It also prevents pests from accumulating. It occurs during the project's development. Rapid problem resolution Testing environments vary with iteration. Feedback is vital. They are always in sync when it comes to project planning and quality assurance. TDD is a way of building software that is Testing occurs throughout Agile implementation. Adaptable to change. Agile Testing Flaws: Regular communication between testers and developers is required. The organization is vague. Only senior software engineers can make significant decisions in team meetings.

**Waterfall Assessment** - Based on Waterfall software development, Waterfall testing is a testing approach. The term "waterfall" refers to a series of stages in which the output of one becomes the input for the next. The waterfall technique is a linear-sequential life cycle model. Agile testing, as the name implies, is used in conjunction with agile development. So, waterfall testing is utilized in waterfall construction. Waterfall Testing Gains Well-structured and recorded testing technique Suitable for all project sizes. The project's features are all delivered simultaneously. They don't have to constantly converse. They stand alone. Easy to use. The Waterfall method contains a set of deliverables and an evaluation mechanism for each level. It's easy to swap teams or shifts. Cons of Waterfall Testing It's a lot of paperwork. The rules are stringent. It's not appropriate for big projects. Testing occurs at the project’s end. Bugs are sometimes easier to deal with in advance. It's less effective than Agile if the client's requirements are vague.

# Section 3.1: Evaluating Efficiency of Metrics

Identifying the metrics consumer is the first step in software engineering monitoring. Our data is useless if it isn't pipelined. Metric consumers include functional managers, project managers, software engineers, testers, specialists, and clients/users. No measure should be generated without a consumer. Metrics are time and resource intensive to gather, report, and assess. Customers' information demands should always drive the metrics program. Otherwise, we risk having a useless product and a useless program. Identifying potential consumers and including them early in the metric creation process increases success rates.

The reasons for metric importance will ultimately guide our development method. We must make wise choices to achieve our internal and external goals. It's challenging to maintain coding standards while developing quickly. On-time delivery of a polished, watertight software system and a comprehensive product is difficult. Trade-offs must be considered, as well as critical choices such as which criteria will be used to grade successful engineering. Managers can analyse productivity using two methods: Size-related metrics show the outcomes of an activity. Consider the source code lines written. Function-related metrics show the quantity of relevant functionality shipped over time. The most common waterfall software metrics are function and application points, while story points are the most common agile metrics. Your chosen productivity indicators should be consistent, meaning they should be auditable so others can verify their feasibility, available so we can compare them, and repeatable so various groups of users get the same result. I'm going to look at the big picture instead of individual stats. Let's look at some different metric groups and see what works. Because we can't understand how different measurements interact, discussing one at a time is worthless.

Now is the time to focus on primary development metrics. The first procedure is formal code metrics. Caution is advised while using this simple metric. The number of lines of code written/edited by a developer is not a reliable indicator of efficiency. However, these metrics should not be utilized as the main basis for monitoring and evaluating software engineering. This is useful information, but it should not be used to make decisions. We can utilize GitHub analytics and progression tests to measure this. We can use task time to see how much code is rewritten.

Code churn, assignment scope, efficiency, and active days are all examples of developer productivity measures. These indications may help you assess a software developer's time and effort. This is an essential fundamental / base measure. This is important for both developers and managers. These indicators are vital because they show us what has been done, what is being done, and what remains to be done. This type of data can be collected using GitHub. I also enjoy looking at general statistics like log-in time, break time, production time, and research time. Keeping us informed about environmental events and

trends. I won't go into how to discern if an engineer is actually working on something or just playing Candy Crush. It would take time away from discussing other interesting measurements. These indications can help us identify present bottlenecks. If we notice that section A is spending a lot of time researching while section B is accelerating, we should meet and discuss rearranging our resources to help section A catch up. These measures can also be used to gauge the final product's progress. Consider what's going on in the work-from-home environment as a result of the coronavirus. Pretend we finished our assignment ahead of schedule before the coronavirus hit. Seeing that our workforce is working from home, the network team has slipped behind, while the frontend team has finished. If we investigate more, we may find that the network team's testing procedures are too slow for working from home. I hope I was able to convey how we might utilize this type of data to assess software development.

Agile KPIs include lead time, cycle time, velocity/throughput, and spirit burndown. They monitor a team's progress toward deploying working software features. We can't just look at the code written to judge our team's software engineering quality. We must also review our goals and progress. Are we being honest? Can we exhaust our team? Is our current cycle too short, or are iterations published too slowly? These are the questions these metrics seek to address. Errors are costly, therefore collecting data and assessing the process is vital. This metric measures how quickly users can get new features. Using total throughput to median time by issue or status type might help you determine your team's current speed. You'll be able to pinpoint performance bottlenecks and set more specific goals. Throughput is the team's total value-added work output. It's commonly represented by the number of tickets (work units) completed in a certain time. Your throughput statistic should match your business objectives. If your objective is to ship bug-free modules this sprint, you should see a high percentage of defect tickets resolved. In agile scrum, sprint burndown is an essential KPI. The state of work during a sprint is communicated through a burndown report. The team's goal is to always deliver jobs on schedule. Tracking this measure reveals a lot about it: Regular early sprint completion may suggest a lack of planned work for that sprint. Missed sprint deadlines, on the other side, may indicate poor planning or overworked teams. An increase in "remaining values" indicates that the job wasn’t assigned in smaller chunks.

Mean Time to Recover and Mean Time Between Failures are both examples of operational metrics. This looks at how well software works in production and how well it is maintained. Because failures are unavoidable in software development, keeping track of them is essential. It can uncover software development faults. Software engineering includes testing and handling failures; thus, we should examine this process as thoroughly as we do software creation. There's no point in designing software systems if we don't have a backup and can't restore them. It's easy to see how Google prioritizes operational metrics. Site Reliability

Engineers (SREs) work at Google to keep Google search etc up and running. DiRT (disaster recovery testing) is a team-run simulated conflict on Google's infrastructure. This "war" includes everything from breaking water pipes to stealing server disks—whatever it takes to bring down the system. The data centre attacks aren't genuine, but they look real. "We've grown in our willingness to disrupt to ensure everything works," the SRE Managers add. "It's my responsibility to create big tests that reveal issues," Krishan explains. Based on a false attack, they may determine what's working and what needs improvement.

Test metrics include code coverage, automated test coverage, and production defects. Measuring thoroughness of testing should be connected to software quality. Finally, the team/management decides the test metrics priority. Because we're engineers, I've found test metrics to be the most intuitive. Test metrics are usually short and easy to extract important data from. We can set up automated testing servers and backup servers in case several tests fail.
Customer satisfaction measures include Net Promoter, Customer Effort, and Customer Satisfaction Score. The pinnacle of customer pleasure with the software and supplier. There are many examples of wildly popular bad software. Despite its shortcomings, Windows still commands 72.98 percent of the tablet, desktop, and console OS markets. In the end, client satisfaction is what makes or destroys software; thus, we must focus on this part of software engineering.

# Section 3.2: Collating Metrics

Collating metrics is very important. Software metrics are used to assess the current state of a product or process, identify areas for improvement, and forecast future performance. Software development managers are trying to improve ROI, manage workloads, eliminate overtime, and cut costs. These objectives can be met by educating the entire firm about complex software development activities. Metrics are important for debugging, quality assurance, performance, management, and cost assessment. Software development teams commonly get goals as software metrics. As a result, the focus switches to minimizing code lines. The reported bug count is decreasing. Increase the number of software iterations. Increasing the pace of completion.

To collate metrics, you must make software metrics helpful to the software development team so they're able to work more efficiently. Measuring and analysing code doesn't have to be difficult or time-consuming. Software metrics should cover critical qualities. They should be simple to calculate. Clear and consistent (objective). Use standard measurement units. Regardless of coding language. It's easy to adjust. It is easy and cheap to get. Testing for accuracy and reliability is possible. This is vital for high-quality software development. To avoid this, software development teams and management should focus on software KPIs that help provide viable solutions to customers.

It may be used for group meetings or check-ins, as well as time tracking and work completion charts (suited for either agile or waterfall development). Of course, covid-19 has modified our methods of gathering information due to new goals. Their purpose is to foresee trends and avoid problems. GitHub is a popular but limited efficiency tracking application. Rather, it monitors total activity. We can learn more about what's going on by collecting more measurements. If health data is to be collected, we can buy fit bits and apple watches that send data back to algorithms that recognize stress and dangerous behaviours. We can also collect market data to identify trends and analyse markets. The platforms you use depend on the issues you're seeking to solve.

"Follow the trends, not the statistics," we must decide which metrics to use. When should we measure? Yes, always. The benefits of gathering measurements are too strong to ignore. The question is where to collect metrics. The simple is also Software metrics are appealing to management because they simplify complex operations. And such numbers are easy to compare. So, achieving a software metric goal is trivial. So, if that quantity isn't met, software development teams know they must work harder. These simple goals don't tell us nearly as much about software measurements. It's the general trend that counts. The process is better understood by examining the trend line's direction and speed. Trends will also show the effect of process changes on progress. I.e., The Hawthorne Effect occurs when people become aware that they are being watched and their behaviour alters as a result.

Sadly, not achieving the aim can be considered as a failure. A trend line, on the other hand, displays progress toward a goal and gives incentive and guidance.

This is a great topic that can help our teams perform better. We evaluate software quality in terms of reliability, maintainability, testability, portability, reusability, and security. We can automate functional, unit, and system tests. We can use the formula productivity = function points implemented/person months. The function point would be a “unit of measurement" for a team's/functionality. developer's This is an enticing measure of productivity because we’re assessing the return on development time. It also considers the end user/consumer. This data can be fed into an AI machine to save debugging time and speed up the adoption of new technology. One example is Facebook's SapFix AI hybrid tool. SapFix develops bug fixes and delivers these fixes to engineers so they can review them and decide on production deployment. SapFix helped deploy robust code modifications to millions of Facebook users on Androids. This is the first time that artificial intelligence-powered testing and debugging tools have been deployed at this scale in production. Following that, the tool evaluates recommended remedies for three issues: If so, is there a compilation error, and does the debugging trigger further crashes? This technology's applications can make software speedier and more responsive. AIs can also be utilized to figure out more sensitive information, like someone's job satisfaction, or their heart rate via a camera.

# Section 4: Ethics

**Good vs. Bad**

The ethics of it all is a highly debated topic. Personally, I think it's fine to measure work with care and knowledge. We'd lose control over development if we didn't monitor work completion. A similar use of monitoring is for symptom-based alerting. But there's a line to avoid. To measure software engineering's efficiency, we must appreciate it. When evaluating software engineers, consider their consent. Software engineers will be productive if they believe they are constantly being watched and their every move is being recorded. Instead, we will strangle growth by being afraid to recommend radical changes. We must first check the fundamentals. I find it hilarious that struggling companies employ illogical metrics to measure software engineer productivity. They penalize developers who don't spend X hours coding, despite the fact their obligations might be completed already. So long as the developer understands what metrics are being collected and how they will be used, there should be no difficulties. Rather than punishing our resources, metrics should be used to improve them.

**Modern Perspective**

Finally, we'll discuss a modern ethical approach. People's choices of matrices are rapidly expanding. This can be good or harmful. Some companies, for example, track software developers' health to help HR make choices. When creating health initiatives and programs, HR professionals must include all employees, just as they must always consider people before numbers. What is the best course of action for the entire workforce? We can learn a lot from metrics, but not everything about your staff. Wellness programs can sometimes be difficult to follow. The main issue is whether companies can require employees to participate in specific wellness programs. Yes, tracking health in 2022 will be easy and valuable, but we must be clear about how we will utilize it. For example, the Australian government has made tracking a serious offence by increasing the maximum fine from AU$126,000 to AU$315,000 for individuals, and banning commercial health insurers from accessing health or de-identified data. Employers in Australia who collect health data cannot fire, intimidate, or treat employees unfairly based on that data. It's excellent to have proactive discrimination protection in light of these nuances, and we will need more of it if as tracking increases in the future.

## References


https://www.lexology.com/library/detail.aspx?g=8c1d8682-2492-41db-9468-45e1ec09f491

https://www.tsheets.com/gps-survey

https://techbeacon.com/app-dev-testing/9-metrics-can-make-difference-todays-software-development- teams

https://www.sealights.io/software-development-metrics/top-5-software-metrics-to-manage-development-projects-effectively/

https://stackify.com/track-software-metrics/