**Measurable Data**

If the measurement of software engineering is important, then we must think about what it is we can measure. Luckily, due to the use of source control frameworks, software engineers effectively leave a trail of data, that describes their every action. As such, there are a variety of metrics that are used to measure software engineering. I feel that the exploration of some of these metrics, not only highlights how software engineering can be measured but also how difficult it is to quantify an engineer’s contributions.

One of the earliest software metrics is called source lines of code (SLOC). This metric attempts to quantify the effort made by an engineer, by counting the number of lines that the engineer contributed.

In theory, this metric provides a good way to measure the productivity of an engineer, and the size and maintainability of a project. I believe SLOC, however, has fundamental flaws. Firstly, if one wished to infer the performance of an engineer by counting the lines of code they have contributed, what would be considered “good”? One person may argue that a large number of lines would suggest a productive contributor, as they have provided a large body of work to the project. It is, however, true that often a better programmer can write more concise code, which is easier to understand and maintain. Therefore, it might be better to have a smaller line count.

A metric often used in conjunction with SLOC is the measurement of bugs per line of code. This is meant to give an idea of the quality of the code being produced. It is certainly preferable to produce code with fewer bugs, as this will less time and resources will be used to fix them. Relying on these two methods for the measurement of software engineers, however, can lead to counterproductive workplace habits. To boost the number of lines that they have committed, an engineer might intentionally write less concise code than they are capable of. The use of bugs per line of code may also encourage engineers to avoid tackling tricky problems.

SLOC and bugs per line could be seen as the most basic software engineering metrics. They are easy to measure and not too difficult to understand. It is perhaps for these reasons that these metrics were utilised in software engineering’s formative years. The issue with them is that they are not aligned with the general goals of a software engineer; that is to reduce the lines of code, reduce the number of bugs, and reduce the time of completing tasks. There are, due in large part to the quantity of data that software engineers leave behind as they work, several metrics we can use.

Ideally, one would want engineers to be efficient in their work. By efficient, I mean what percentage of their code is productive. This could be measured by balancing the coding output against the code’s longevity. If an engineer has high efficiency, then their code provides monetary value to their business for a longer period.

A useful metric for calculating is called “code churn”. This is a measurement of how many lines of code were added, deleted, or modified in a specified period. If an engineer consistently has a high code churn, that might indicate that engineer is not efficient.

Code churn can give a variety of insights into the progress of a project. In the early stages of a project, there will be a period of prototyping and exploration. This naturally will lead to a high code churn. As a project progresses towards release, one would expect to see the code churn to decline.

When engineers encounter a difficult problem, a large amount of backtracking and exploring will occur. It is almost inevitable that a team will encounter such problems during the course of developing a project. These problems will result in a spike in the code churn.

A high rate of churn could also be caused by unclear requirements and issues surrounding uncertain external stakeholders. These situations can cause a large amount of frustration within engineering teams. If these situations arise, it is important to define concrete requirements.

Probably the most important thing a software engineer can do is to make code contributions. Often, engineers have undergone years of train and acquired university degrees dedicated to providing them with the skills to build and solve difficult conceptual problems. It is for these skills that many companies are willing to pay high salaries.

In reality software engineering involves process overheads. Tasks such as planning, meetings, research, and gathering specifications are essential to the process of engineering and require time. Ideally, however, an engineer would spend as little time as possible on these overheads, and more time contributing to the code base. Active days is a metric that counts the days that an engineer contributes code. This metric allows a team to see the cost of these overheads and ensure that the overheads don’t overburden them.

Another important metric is impact. Impact is a measure of the effect that an engineer’s contribution has on a project. The impact depends on many factors, for example, the amount of code in those changes, the severity of those changes, and the number of files those changes affect. Impact can give an indication of which engineers are more familiar with the different aspects of the project. A contributor who has impacted several files in the project is probably going to have a greater understanding of the entire project than a contributor who has only impacted one file.

**Platforms for Gathering and Analysing Data**

As I have stated, every action that a software engineer makes gets creates data. That means every line of code, every comment, every commit, every pull, and every contribution to a discussion board generates data that can be gathered, stored, and processed. As you can imagine, each engineer generates a huge quantity of data. Knowing this, and knowing that in a company, there could be thousands of engineers, raises an important question, how is it possible to perform computational operations on such a large set of data? I will now discuss how the dawn of cloud computing has made it practically and financially possible to gather and analyse extremely large sets of data. Then I will explore some of the specialised frameworks that have emerged to perform data analytics.

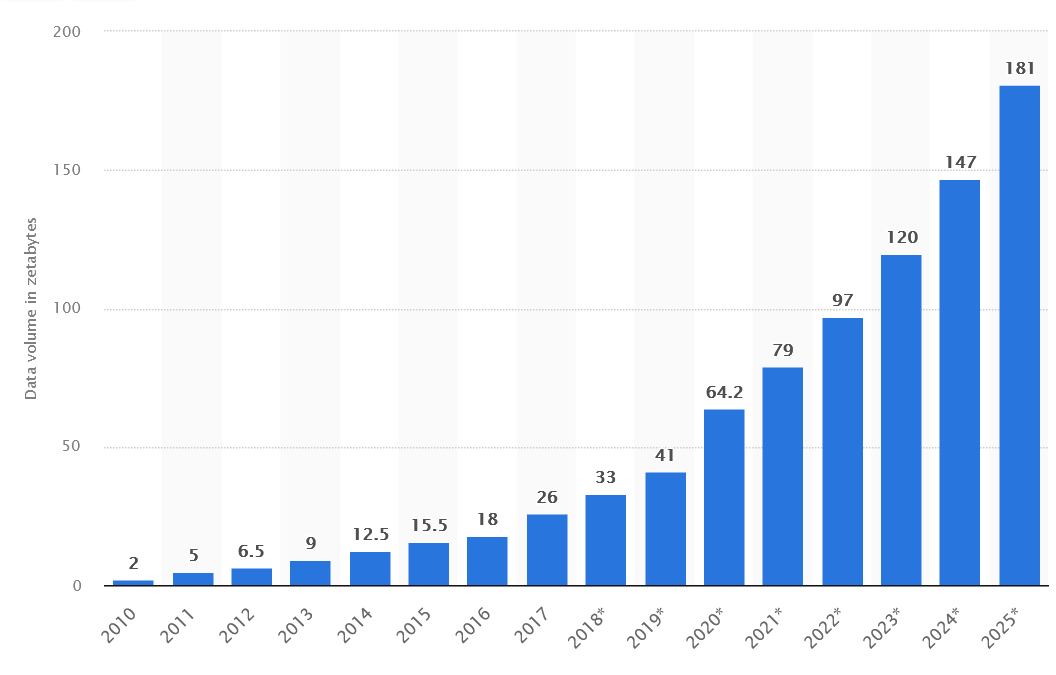
First, I would like to discuss the concept of big data. The amount of data we produce has grown exponentially in the past ten years. In 2010, 2 zettabytes (221 bytes) of data were created, captured, copied, and consumed worldwide. In 2020, however, that figure rose to 64.2 zettabytes. That means, that every person generates 6.185 megabytes of data each second.

Figure : Data generated worldwide from 2010 to its projected figure in 2025. https://www.statista.com/statistics/871513/worldwide-data-created/

There are several different definitions of big data. Wikipedia defines big data as data sets too large to be dealt with by traditional data-processing application software, and its challenges include capturing data, data storage, data analysis, search, sharing, and visualisation. Sam Madden defines big data as data that is too big, too fast, and too hard to be for existing tools to process. In 2001, industry analyst Doug Laney introduced the 3V model to describe the features required for data to be considered big data. The three Vs are:

1. **Volume:** This describes the size of the data. Volume is often considered the defining characteristic of big data. Generally, terabytes or petabytes are considered the benchmarks for big data.
2. **Velocity:** This refers to the rate at which the data is generated or the rate at which it must be processed. When we consider velocity, there are two main types of data processing: batch and stream. Batch processing allows data to be stored over a period of time and processes the data in blocks. This model is suitable in a situation when no real-time processing is required. Stream processing is the opposite. It allows data to be as they arrive, and it is key for real-time processing and data analytics.
3. **Variety:** This refers to the various types of data being produced. Big data is classified into unstructured (audio, video, images, etc.), semi-structured (XML, JSON, etc.), and structured (spreadsheets, relational databases, etc.).

Storing, processing, and analysing big data requires an extraordinary amount of computational power. Using standard, commercially available desktops or laptops to analyse big data would be either impractical or impossible due to the amount of time and energy it would take. Purchasing the hardware required to work with big data is impossible for most businesses due to the high cost.

Cloud computing, however, means it is now possible to work with big data. Cloud computing is the on-demand access, via the internet, to computing resources. These resources could be applications, servers, data storage, development tools, or networking tools. There are three main models of cloud services:

1. **Software-as-a-Service (SaaS):** SaaS is application software that is hosted in the cloud. You can access the application through a web browser, or a desktop client. It is the primary delivery model for most commercial software today.
2. **Platform-as-a-Service (PaaS):** PaaS gives developers on-demand hardware, infrastructure, and development tools for running, developing, and managing applications.
3. **Infrastructure-as-a-Service (IaaS):** IaaS provides access to computing resources, such as servers, networking, and storage over the internet.

Possibly the most important aspect of cloud computing, in a wider context, is that it often follows a pay-as-you-go or subscription pricing model. This means that businesses do not have the high capital expenditure of building their own, bespoke infrastructure to process big data. They only need to use the infrastructure built by cloud service providers and pay for the time and power that they need.

Now that we have explored the possibilities of exploring large quantities of data, I will now discuss some specialised platforms for gathering and processing data in software engineering.

**GitHub**

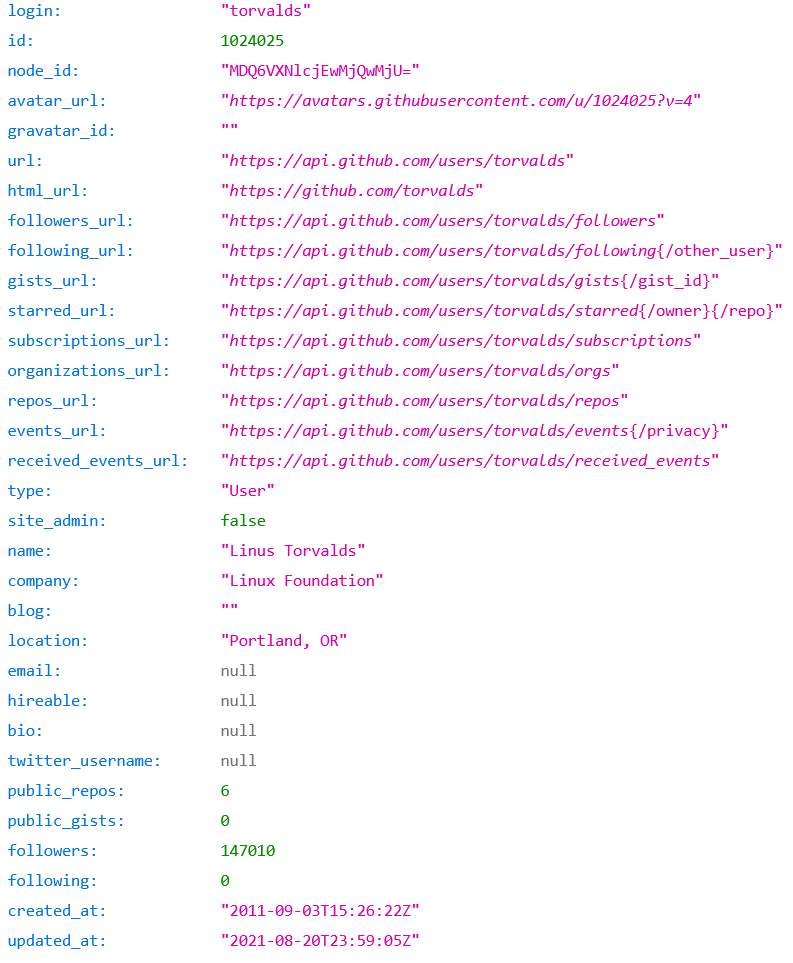
GitHub is an internet hosting service for version control using git, founded in 2008. GitHub provides all the functionality of Git, including commit, push, pull, branch, and merge, as well as feature requests and wikis.

Figure : GitHub API response for the user account of Linus Torvalds

As of 2021, GitHub reports having 73 million developers and more than 200 million repositories. Crucially, GitHub stores a massive amount of data on these developers and repositories and makes this data available to access through its Rest API. Using the API, a company can essentially access the raw data on what each of their engineers is doing. With this data, using languages like Python, they can create bespoke processing and analysing tools to evaluate the performance of their engineers.

**Pluralsight Flow**

Formally GitPrime, Pluralsight Flow is an engineering analytics platform. It is capable of gathering information on commits, pulls, and commits across multiple platforms. Pluralsight believe that their metrics can help engineering teams identify bottlenecks in their workflow, thus increasing the team's efficiency. Flow’s metrics are broken into five categories:

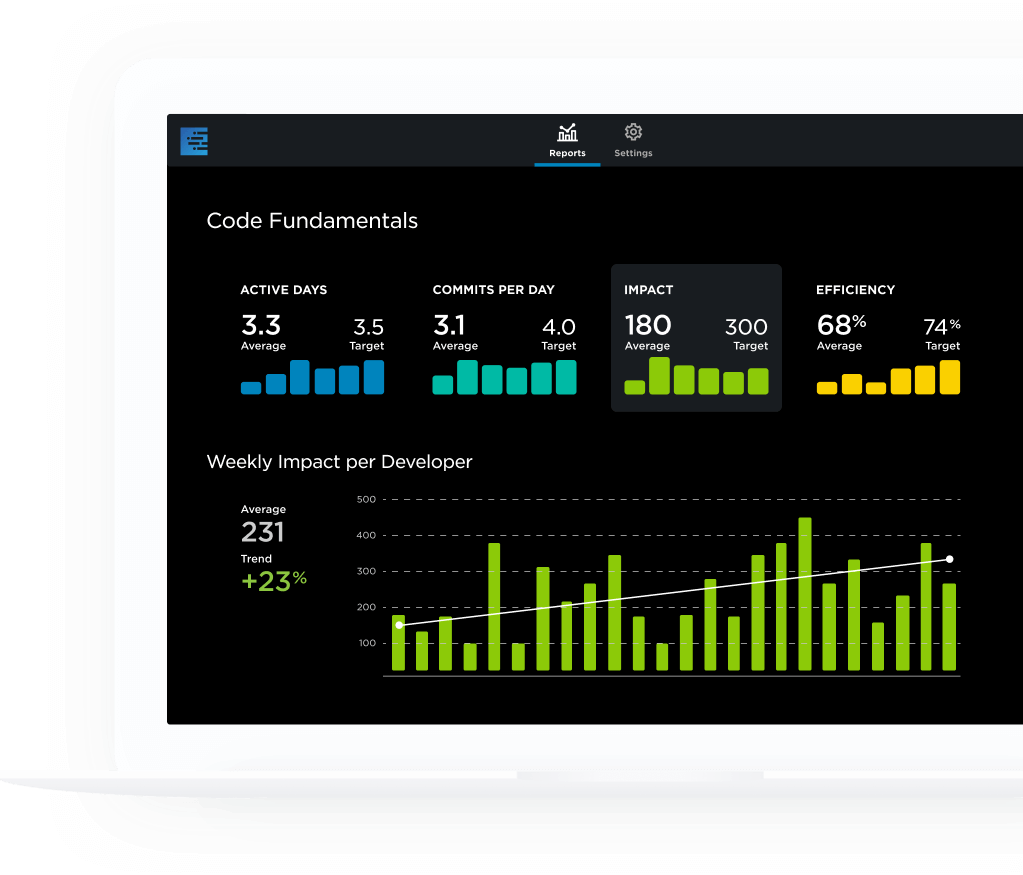
1. Code metrics. These include coding days (a day when a developer contributed code to the project), churn (code which is deleted or rewritten shortly after being written) and impact (the severity of edits to the codebase, as compared to repository history).
2. Submit metrics. These include responsiveness (the time it takes a submitter to respond to a comment on their pull request), comments addressed (percentage of comments to which a submitter responds) and receptiveness (percentage of comments the submitter accepts).
3. Review metrics. These include reaction time (the time it takes for the reviewer to review a pull request), involvement (percentage of pull requests that the reviewer participated in) and influence (ration of follow-on commits made after the reviewer commented).
4. Team collaboration metrics. These include time to resolve (the time it takes to close a pull request), time to first comment (time between when a pull request is opened and the time of the first comment) and follow-on commits (number of code revisions added to a pull request).

Figure 3: Example visualisation from Pluralsight Flow

1. Knowledge sharing metrics. These include sharing index (how information is being shared amongst a team), number of pull requests reviewed (total number of pull requests that were reviewed) and number of users reviewed (total number of submitter users that were reviewed).

**Code Climate**

Code Climate is a web-hosted software that provides analytics and reviews of code. Like Flow, Code Climate works in conjunction with Git. At every pull request, Code Climate analysis the code, considering the complexity, churn, duplication, and coverage, to quantify the quality of the code. It then provides automated comments on the pull requests. These comments are designed to focus review discussions about the code, thus making the team more efficient.



Figure 4: Example output from Code Climate