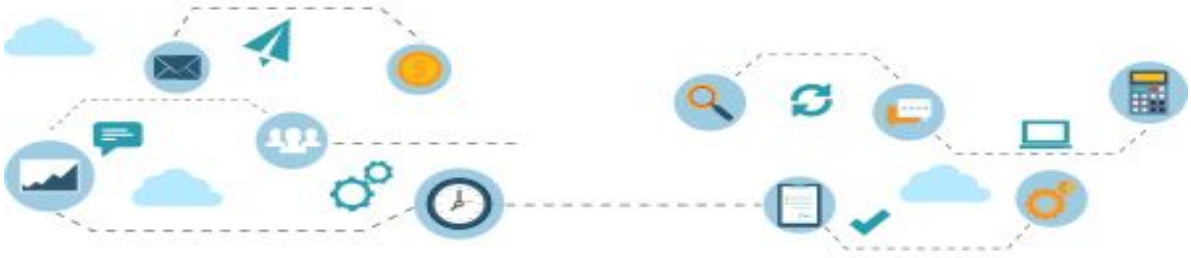
CS3012

Measuring the Software Engineering Process

Patrick Lee - 15325692



Introduction

This report will examine the software engineering process, from the methods of monitoring and recording performance to the ethical implications of imposing such scrutiny. The first chapter will consider the former; specifically how one can mould the process into usable data. I will give an explanation of current procedures and accompany them with a discussion on their merits and potential shortcomings. The second chapter will have a look at the platforms available to carry out the analytics; exploring companies like CodeClimate and delving into the field of Application Release Automation. I will then investigate the algorithms utilised by these platform; lots of the data doesn't require complex manipulation to provide useful information, however, the advancement of machine learning promises to add more complexity. I will conclude with a consideration of

the ethical concerns surrounding these systems, first outlining the standard procedures when monitoring a development task morally and then applying them to the processes discussed for software engineering.

The Software Engineering Process as Measurable Data

In the early days of software the main focus of quality control and performance monitoring was predominantly around testing. As technology advanced, and with it the capabilities of software engineering, the industry began to evolve its processes, with each update skirting around the idea of focusing on the engineers themselves without ever embracing it. Regular inspections of the software were adopted to reasonable success before a new model, the Capability Maturity Model (CMM) , was introduced. This system put management and the resources they granted to the engineers under review and had a similarly positive effect on software quality. It wasn't until the Personal Software Process however, that the programmers themselves were the ones conditioned.

Spawned from minds at Carnegie Mellon University, The PSP's main aim was and is to instill good practices into the software engineers so they can set goals, plan, execute and analyse the development of their task to, not only produce the best solution the first time around, but to also keep track of what worked and what didn't to improve their performance the next time. The key principles focus on planning ahead and preventing issues rather than fixing them later, the process also encourages clarity from employers on their monitoring and that the engineers have pride over their work and are determined to produce the best that they can. But in order to achieve their goals there had to be a way to measure the process in terms of tangible data. For PSP there are three main measures: the size of the program, the time taken for completion and the quality of the program.

Today there are countless ways used to extract data from the software development process, all stemming from the three outlined in PSP and each having their own merits specific to the business involved and the task being worked on. I will consider the different methods under the headings of Code Analysis, Program Quality, Productivity and Security and determine which I believe are the most useful and which don't necessarily provide any great insight. I would like to stress again that each metric is arbitrary and will work better for certain businesses over others.

Code Analysis

It is perhaps the simplest way of converting the software engineering process into measurable data by looking at the actual code, but also perhaps the most unhelpful. Lines of code (LOC), number of bugs per developer, number of tests passed, external libraries used etc.. are all statistics that look good on paper but in reality are somewhat misleading. These numbers provide useful information on the overall size of a project, a more

important metric, but offer little individually that a team could use to improve for the next task. The point being that a program with less LOC or more bugs at a given time isn't necessarily better than a very long one completed without a hitch. Each program is different and so is each developer. Rather than looking at the amount of bugs, a better use would be to monitor the time it took to remedy them and then to note what steps could have been taken in advance to prevent the issue.

This information should be used to analyse trends and keep a record of certain tasks so long as the completed program is suitable. It should never be the measure of the success of the project as a whole. The most important factor is ultimately the final product and the satisfaction of the user. The key metric this information provides is the size of the project. By understanding the scale of the task, the team can then give better estimations for the time needed for upcoming tasks, have a better understanding of where to split the project into subtasks and possibly identify future issues earlier and then accommodate for them in advance.

Program Quality

Rather than measuring the quantity of the code, it is more beneficial to examine the quality. The number of bugs or defects gives a good indication, but crucially at the final product and not necessarily during production. A program riddled with defects is obviously poor quality code but to identify areas that could be improved when the code is above a minimum standard you have to consider other metrics. A better determiner in my opinion is also a reasonably arbitrary measurement. Good quality code should be efficient, readable and compatible with all other work in the team, but these characteristic are hard to convert to scalable data. Instead the measurement boils down to time. If the code meets the criteria, then the time finishing subtasks should be relatively short and in line with what was planned, the work integrating one engineer's work with a colleagues should be seamless and any technical debt should be kept to a minimum. If too much time is spent on any of these inevitable steps then the code is not up to scratch.

These issues can also be avoided with better practices. Technical debt for example, concerns implementing easy, working code quickly, rather than the best solution; which leads to reformatting to accommodate additional functionality and difficulty with integrating new features. By recording each instance, and the time that was lost dealing with it, you will be better equipped to avoid the problem the next time. There are also a few approaches some use to, not only monitor but, maintain the quality of the code to a set standard. Implementing peer reviews as well as frequent spot checks makes sure the code is suitable as well as utilising pair programming to give each task a second pair of eyes.

Productivity

To be able represent the productivity of a software engineering process as tangible data, it is perhaps unsurprising that again time should be the main focus. Many companies use an Agile system for their project management and within that a framework called Scrum. The main metrics for these methodologies are:

Lead Time: The time taken from an idea's inception to 'physical' completion.

Cycle Time: The time taken to alter your software and implement that change in production. It is included in lead time.

Team Velocity: The number of completed "units" of software during a "sprint".

Normally a project is broken down into a "Product Backlog", which contains a range of subtasks like new features and bug fixes, and a selection of the pieces are then scheduled to be completed in a sprint; which can last a couple of weeks. These are useful metrics to have when planning another task or project as they give a good insight to the time the team and it's individual's need to resolve certain problems. The group can then give better estimations to future clients and dissuade any undue stress as a result.

These measures for productivity can also be beneficial when monitoring a group's communication. By recording the time needed for each section, while noting what took place and the reasons for any delay, it allows a company to see in what areas the team can be improved. If a large portion of time is spent on planning and organising or fixing errors due to a lack of understanding or conveying of information, then the team knows that they need to improve their communication.

Security

A crucial area that can offer a lot of useful metrics but also perhaps doesn't receive as much attention is security. It is heavily linked to code quality as poorly written code can make a program vulnerable due to its unpredictability. Bugs and defects present potential threats and provide more options for malicious types to cause problems. Also if code is difficult to understand can be hard to fix an issue quickly which has the potential to be detrimental in the case of a breach. These introduce the measurements that can be calculated:

Endpoint incidents - The amount of viruses experienced by various users over a certain time.

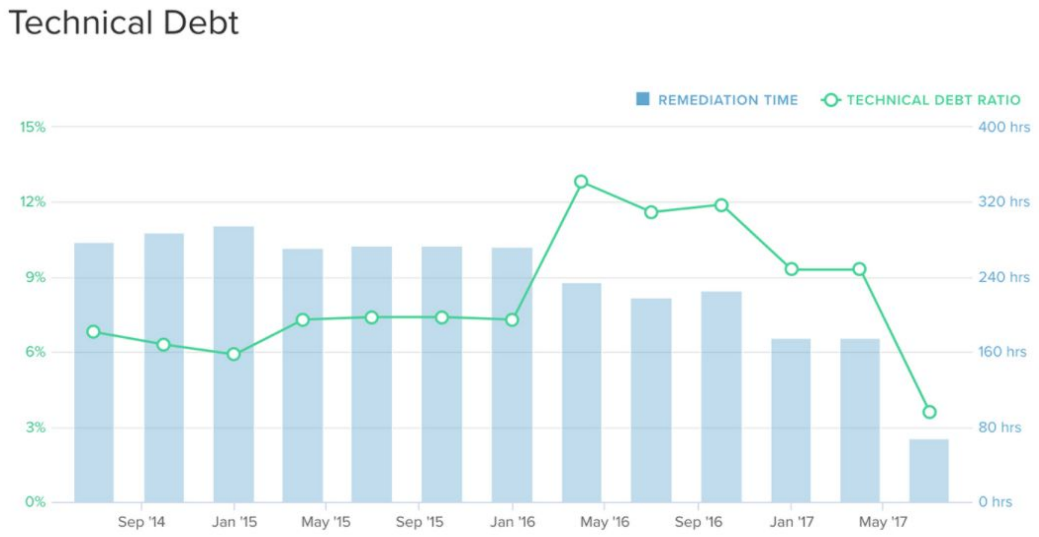
Mean Time To Repair - or MTTR concerns the time between the discovery of a security issue and the implementation of a fix. The mean time of a collection of these instances provides the calculation. [1]

We can see there are many different methods of measuring the software engineering process and these metrics can give a useful insight into a team's ability to work together and produce a working product. This data can then be used effectively to improve the planning and organisation of the team for the next task. Each company, task and individual developer is different, it is just a matter of choosing the metrics that are most suitable.

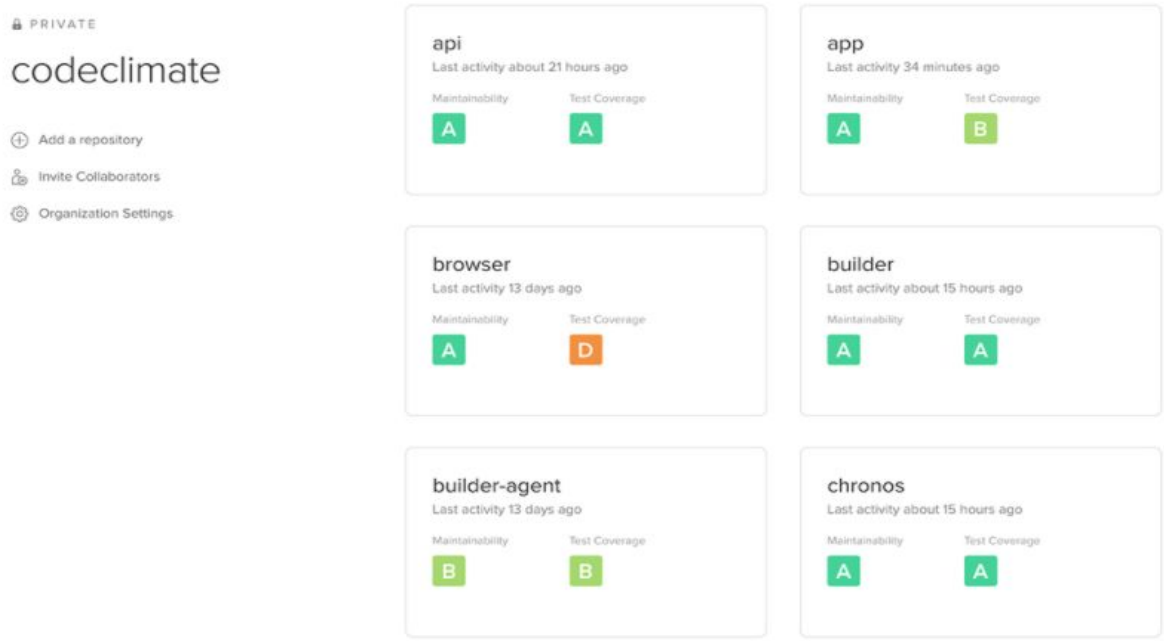
Platforms Available

There is a plethora of tools available for measuring the discussed metrics of a software development process. Most offer rigorous testing and code coverage, while also identifying security risks. There is also a burgeoning field within the process of Application Release Automation which attempts to smooth over any integration issues across environments as well. Utilising software monitoring systems for project management is swiftly becoming the norm and with it the number of platforms available. I will look at a few of the most popular currently and discuss the features they offer.

Among the most popular companies providing data collection and instant statistical illustrations are CodeClimate, Codebeat and Codacy. Rather unsurprisingly they provide nigh obsessive feedback on your program's code. They scrutinise each line and identify anything from security risks or code duplication to whether or not they like your style of writing code. This is where the line by line facts and figures spawn, through thorough testing, and returns grades on code coverage and maintainability which is normally sent back in the form of a report or diagrams.

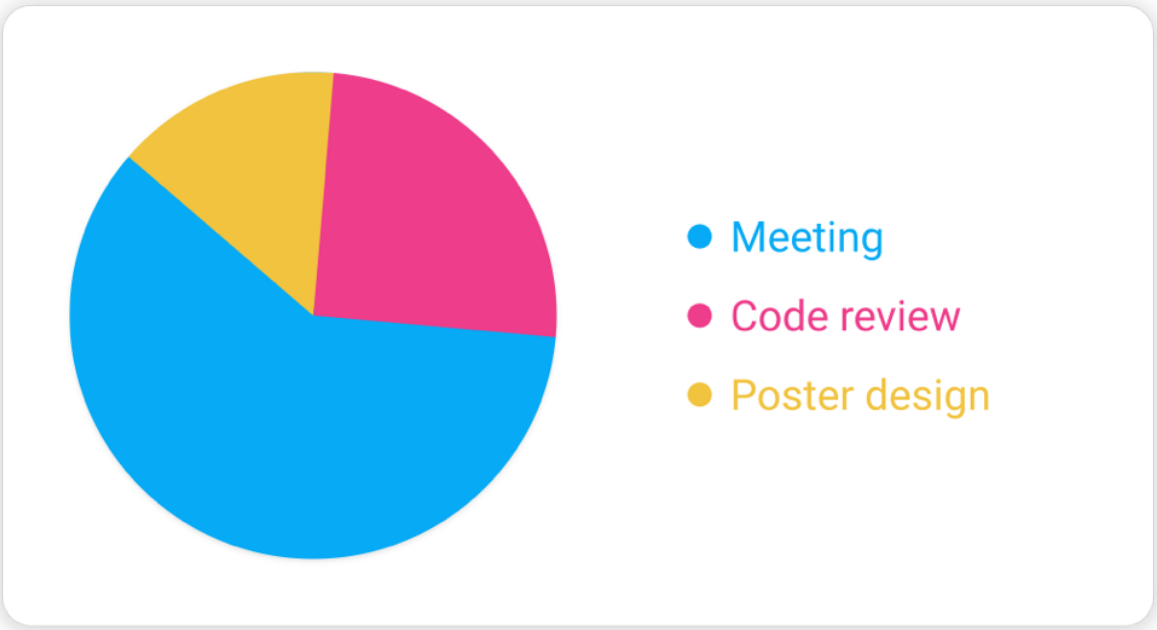


CodeClimate, for example, monitors technical debt and the time spent fixing it. They also offer a security dashboard which gives a list of the programs weak points, including when they were introduced and also recommendations on how to resolve them. The code reviews they form into reports are often tailored to the team's specifications as the data that's collected for coverage and maintainability is entirely configurable.



Codacy have a similar approach, alerting the user to instances of code duplication and giving feedback on compatibility. They also provide data on security issues and vulnerabilities, such as errors and defects. Other programs like Codebeat tend to stray away from the group as a whole and choose instead to focus on the programmers. They have less emphasis on grades and graphs and more on promoting the skills of the individual through a strive for continued excellence.

These platforms provide the tools for code analysis and giving the programmer instant feedback and data on their program as soon as possible. Where a lot of the key metrics I mentioned in the previous section are centered around timing which doesn't necessarily need a fully kitted platform, there are services available to manage timing. Toggl, and Jira as an extension, are popular tools for measuring the time taken on particular tasks. They can be installed as a simple Chrome extension and used like any other timer. The tool however logs the information and can provide feedback on how much effort is being dedicated to each area.



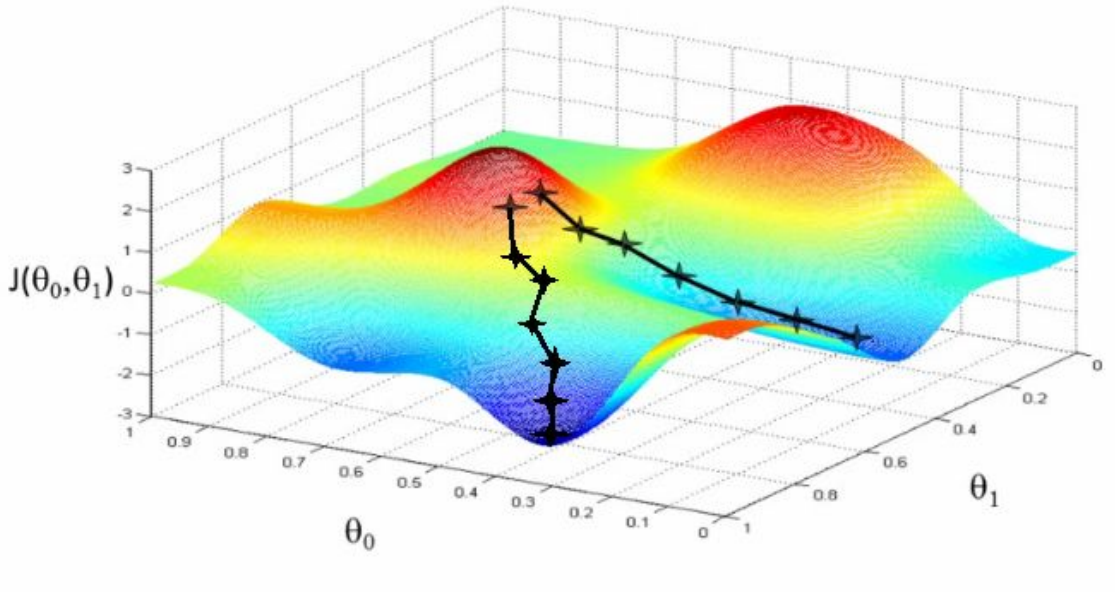
A team can then record their sprint times, how long their meetings are taking or whether certain areas like planning or code checks are being neglected and then use that data to altered their approach in the future if needs be. The platform also provides a simple, handy tool that allows you to create projects and monitor their budgets or deadlines.

An interesting area within project management is Application Release Automation or ARA. This tries to automate the development process to allow companies to deploy new products or add features quickly and seamlessly while also giving feedback on the program's scalability, resilience and security. This is achieved through services in automating deployment, providing management and modelling for the various environments the team are using and also coordination for release of the product.
The platforms improve the quality of the group's code, speeds up the completion time and encourages them to use the best practices through monitoring and managing workflow. A lot of companies are beginning to put a lot of development into this with IBM positioned currently at the forefront. It is thought that by 2020, 50% of global enterprises will be utilising an ARA solution.

Algorithms Utilised

WIth a lot of this data it is usually quite trivial to organise it into related areas and derive obvious conclusions from it and the algorithmic approaches rarely stray far from basic formulae. However, Computational Intelligence, a more rudimentary version of Artificial Intelligence, through machine learning is becoming markedly more potent, and its usage ever more ubiquitous.

Machine learning involves ‘training’ a program to analyse data to give some useful information. The training can be achieved through supervised or unsupervised learning. The former concerns a scenario where you have a large selection of inputs with known matching outputs and through mathematical algorithms, such as gradient descent, can teach the computer to derive a function that fits the given inputs and outputs. Gradient descent for example, is a process where the computer ‘moves’ down a slope of cost values until it reaches a local or global optima. This is the point where the outputs of the function deviate the least from the real, correct outputs. Unsupervised learning however is where you hold a lot of data but don’t know what the answer or correlation is, if there is any. So you give it to a program, which analyses any patterns that could prove useful.



Gradient Descent for a cost function.

This promises to be a useful tool when dealing with all the measurements that can be accumulated from the software development process, particularly with a large project that would yield a lot of data. An interesting idea at the moment is the notion of Software 2.0. Software 1.0 refers the the classic stack utilised in programming languages such as C and Java etc. that gives a computer explicit instructions on what line to run next. Software 2.0 concerns neural networks that are defined in weights. The design makes use of machine learning tools, instead of writing each function, a programmer clarifies the constraints of the program and the network determines the appropriate function through processes like gradient descent.

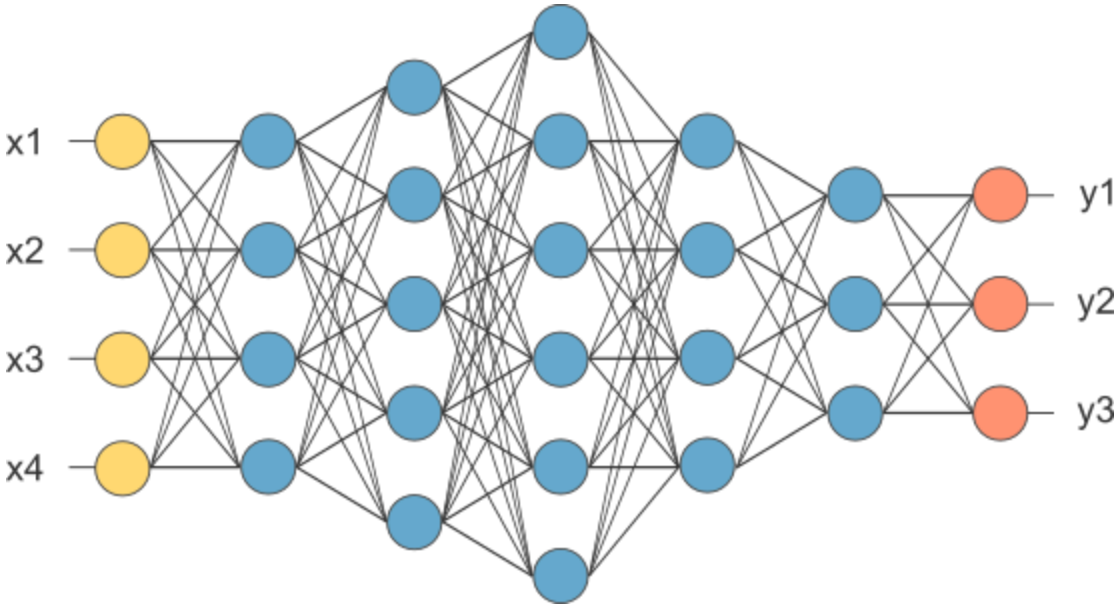


Illustration of the layout of a Neural Network.

This is very handy as it is easy, for large firms in particular, to gather mountains of data that prove too vast for mere humans to spot the correlations, let alone write a function that governs them. It is relatively easy to look at the data of one engineer, such as LOC or number of bugs, and come to conclusions, but it can be time consuming. Neural networks are able to identify any trends and can also handle the large quantity in no time. They are then very useful in areas like speech recognition or speech synthesis, as a sizeable portion of the data can be noise, or unnecessary data, and they can cut through this to pinpoint the result you need. Often, regardless of the metrics being used, the data collection is imperfect as a project will never be completed seamlessly and this introduces the noise. There is also scope for them to be implemented in machine translation, where a source language can be translated to a select target language. This could allow a company to accumulate lots of compatibility data for monitoring code quality.

There are lot of benefits to neural networks, such as constant runtimes and memory use, and a few key ones for working over this data. A neural network is very agile as it consists of lots of channels which operate on whatever arbitrary amount of data you wish. If you halve the channels, the speed doubles but at the cost of diminished accuracy. It's very useful if a business expands and therefore takes on a lot more data. If you simply add more channels and retrain the system it will be able to cope with the added workload. Another benefit concerns the makeup of the program. The software can generally be broken down into modules that communicate over APIs or other publicly accessible functions. This allows the network to accommodate any new features that a company may want to monitor with very little hassle.

The main issue with Software 2.0 and neural networks is the ignorance involved. A lot of the software works properly, however, the functions or underlying code behind them is unclear. The result is a system that gives perfect results that we don't quite understand. The danger of this is that the system can fail embarrassingly, such as Microsoft's trialed Twitter bot, that analysed all the language and ideas shared by the platforms users and ultimately adopted a lexicon with unedifying profanity and a distasteful sentiment. A program can also 'fall silent', this is the case where, in the training phase, the program has garnered certain biases with its output and, although the answers seem correct, gives the wrong result. This is a point where nefarious individuals can exploit by using 'adversarial examples', or incorrect data that manipulates the system to adopt biases.

There are many different ways of computing the data obtained from a software engineering process, it is the use of machine learning however, that could be the most beneficial. The Georg-August-University of Gottingen's Computer Science department are currently researching the use of machine learning for software engineering metrics. They were able to detect *feature freeze* points, this is where all work towards new features ceases and focus is changed to bug fixing and UI improvements, on open source data. They achieved this using the k_means clustering algorithm that took the LOC and number of bugs as metric.

Ethical Concerns

The ethical concerns for monitoring a software engineering process are similar to any ethical considerations one should take when recording the work of employees. A few basic strategies should be followed, which include:

Clarity in Monitoring - Employees should be under no illusions of the level of scrutiny they are under, whether lenient or strict. Normally this is agreed with a consent form, but the terms should be reasonable so the employee doesn't feel like their privacy is being invaded.

Aims Congruity - The purpose for each monitoring process should be entirely transparent and should not blur the lines with tracking. Higher level personnel should then accept similar scrutiny for possible corporate violations; as much clarity as possible is important.

No Personal Data - Information like race, political opinions or sexual orientation should not be under the remit of the monitoring and is generally considered illegal. Keystrokes should be off-limits as well as passwords for example could be identified.

After identifying the key areas and applying them to the measures taken from a software project, it becomes clear that a lot of the monitoring I have discussed is valid:

Code. A lot of the data taken is related to the code written by the engineer, which is part of the team project and shouldn't contain any private information of the individual. The company's aims to be able to improve output quality and quantity through monitoring the development process, and here specifically analysing the code, are relatively straightforward. Therefore, no real ethical issues exists surrounding these metrics in my opinion, unless perhaps some of the tools used by the engineer contain sensitive information. But that is there responsibility and If they have been given clarity on what is being looked at then I don't see there being any morality concerns.

Program Quality. The metrics here concern the bugs in a program and the time taken to fix them along with the time needed to complete tasks and merge code. Unless the engineers take exception to the scrutiny and consider the standards unfair and feel they are being mistreat, then it is hard to see where any issues could arise. The measurements here would be fairly routine and it is the least an employee should expect when having a group's performance checked.

Productivity. Here is where the issue of ethics becomes a touch thornier. Measuring the time taken for meetings, subtasks and overall project completion falls into the region of standard operating procedure, but documenting meetings for instance, could provoke some ethical considerations. Sometimes during a project the team may need a push or harsh truths to keep working effectively. Employees should then feel comfortable expressing their opinions, within reason of course, without having to worry about serious ramifications to their position. If meetings are monitored and scrutinised so closely then the staff may feel like they are being conditioned and then become apprehensive about speaking out if they are being unfairly treated. The possible damages can be a distrustful atmosphere around the workplace, which can lead to employees leaving, and a plateau or decline in productivity.

The important caveat with monitoring is that it is conducted appropriately. This means that all the information being gathered is strictly related to the task or project and not the employee's personal activities. If the monitoring extended, for example, to tracking the internet usage of a staff member, then it is possible to extract personal data inadvertently, such as sexual orientation or political views. The action itself is legal but this scenario makes it ethically wrong. The key is to only monitor that which is open source, like the source files in a programming task. Another crucial factor is that the information should be stored properly and not shared freely. It is vital that an employee is certain that all the data accumulated is protected from third parties, or else they would feel violated. Only the manager of the project should have ease of access to the information.

Conclusion

To summarise, the process of measuring software engineering development is entirely arbitrary. There are countless variables at play and, fortunately, countless tools and methods to accommodate all of them. Some companies or teams may be very code-centric and obsessed with writing the most efficient and clean program possible. They would be more keen on platforms like CodeClimate, that can give instant feedback to allow them to implement instant improvements. Other firms may be more interested in quantity, striving to complete projects in the shortest amount of time. Time is then their concern and reducing the amount spent on nearly every phase, from coordinating to deployment. An advantage is that, regardless of a groups preferences, they don't have to neglect other methods entirely. It is relatively simple to implement many means of monitoring simultaneously while not losing focus on their principle metrics. I hope I have also highlighted the flexibility concerning these measurements. There is a lot of room for maneuver, irrespective of the methods used, which is only increasing with the new developments in software. Sooner or later a lot of the steps in a development task will be automated and with it the metrics involved.

[1]
https://techbeacon.com/9-metrics-can-make-difference-todays-software-development-teams