University of Dublin

TRINITY COLLEGE

Software Repository Code Assessment for Team and Individual Socio-Metric Performance Prediction

Edmond Michael O' Flynn

B.A.I. Engineering

Final Year Project May 2017

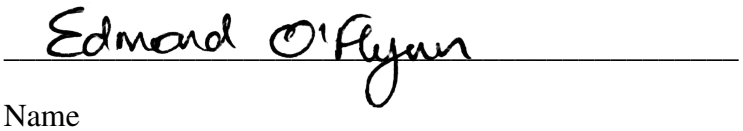
Supervisor: Professor Stephen Barrett

School of Computer Science and Statistics

O'Reilly Institute, Trinity College, Dublin 2, Ireland

DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university



Name

05/05/2017

Date

ACKNOWLEDGEMENTS

Many thanks to Professor Stephen Barrett for the help, support, and many interesting meetings over the course of this project. You are a great wealth of information and an inspiration to becoming a software engineer.

To my parents, Olive and Michael, I can't express enough thanks for giving me the opportunities that you never had. I wouldn't be in my position today if it hadn't been for your sacrifices.

To my sister, Deirdre, thanks for everything over from childhood to today and beyond. You helped me adjust to living in a new city and have always been there for me at every turn.

As gach rud, táim fíor-bhíoch daoibh. Go raibh maith agaibh uilig ó mo chroí istigh.

Table of Contents

List of Figures

ABSTRACT

Software development has captured the world of today with its prevalence in everyday life. With advancements in computing, a change has been brought about in how we use electronics to facilitate everyday tasks in our lives. This bring about the question of the role of the software engineer and its deep importance in developing and maintaining our domain of existence. Software engineering pertains to the design, development, and testing of system applications in a systematic approach. However, there is a large dynamic spectrum on which a skillset of an engineer can lie, where these skills can become more developed and honed with time and exposure.

To understand how large-scale software engineering projects come into fruition, we must first look at the path of evolution taken from their initial conception of the project to their current state. By using a revision control system such as Git, it's possible to iterate through commit snapshots to analyse the series of steps taken to arrive at the current stage of development. It is these snapshots that allows for a temporal axis to be generated, on which observable shifts and patterns can be reviewed and used to analytically generate meaningful metrics. These metrics can show in depth how members of a team interact with the source code of a repository, and thereby allows for metrics pertaining to technical, social, and organisational kinks and mannerisms to be observed more clearly on a graph over time.

This project aims to develop a minimally viable suite of tools to solve the challenge of being unable to explore and generate data sets for given projects. Expanding on this provides a systematic approach to software engineering with the means to allow for exploration of the cohesion of contributions in open-source software repositories, and generate metrics accordingly to demonstrate the differentiation in quality of individuals' work towards a shared goal of creating software. The hope is to more clearly examine and observe the given effects of varying stimuli on the individual's contribution level.

The intended end-user for this platform is be a data scientist who is interested in generating metrics about the potential effects and states of varying methodologies and practices of software development paradigms. The net result of this is the integration into continuous systems for socio-metric performance disposition to analytically see over a large set of data the most optimal way for developers to work on a project in tandem. The results of the project can be summarised as to have shown a clear starting point for metrics observed from the system to infer certain behaviours result from given engagements. Future progress may uncover interesting shifts in modern software development cycles in the scope of interesting properties associated with engineering processes across individual and team efforts. Overall, there is a clear potential for large open-source software repositories to follow distinct stages of maturity as per complexity, with varying roles of contribution ranging from a major addition of features, to those of less frequent additions.

The context of this is to highlight the effects of a set of contributions over time, and the overall effects that large groups of software engineers to varying degrees of competency can have on the overall scope of the project and the complexity variations that ensue. With the consideration of modern practices of open-sourcing software repositories, eliciting the help of many has become a regular practice where anyone can donate progression in the form of a feature, bug fix, or code review through a pull request has become the norm. This has led to a certain culture of working with large groups of other people in a purely virtual manner – with interactions stretching to logging and assigning issues, and working together to create agnostic code that works together in a seamless mesh.

INTRO

MOTIVATION

Software design is defined as "all the activity involved in conceptualizing, framing, implementing, commissioning, and ultimately modifying complex systems". (Freeman & Hart, 2004) As it is built upon making tools for others, it therefore seems ironic that gauging this sense of progression is difficult to evaluate empirically through analysis software for engineers themselves. In organisations, departments have goals to achieve through some form of metrics – teams dealing with sales must sell a certain quantity of stock, leaders have goals set out, but software engineering teams have a general goal of shipping software by a certain deadline. It then seems that this process isn't transparent in terms of metrics. What level of quality defines shippable? How should a team deal with technical debt?

Progression in software development is not only difficult to empirically gauge, but is partially based on elements of human metrics. The goal of everyone in the team is to foster progression and experience a sense of getting closer to reaching a milestone of development. Leaders within fields of software engineering have the difficult task of discovering the current state of play for developers in the team. Simple questions relating strives forward being made can be notoriously difficult to answer – what defines enough progress? What differentiates between progress from narratives of greenhorn engineers to the field, to those well-seasoned?

The initial purpose of this report is to outline the steps needed to provide a tool-chain of deployable infrastructure to provide a method of acquiring, generating, and exploring data sets. The severe lack of tools for leveraging as a method of exploration within this field led initially to the challenging goals of creating a semi-automated process across a system topology to readily acquire open-source projects from GitHub and generate data sets for the consumption by data visualisation tools. The infrastructure must then run processes for generating original data sets of useful metrics across the repositories gathered, so that information may be harvested and providing the basis for any investigations into the analysis of individual and team based software repository socio-metric code analysis.

Building on the challenging task of providing a basis infrastructure to generate the required data sets, the next goal of this report is to explore the area of software code repository analysis for human metrics in relation to code over time as developers contribute to a code base. As code is a representation of the contributions of an engineer, there is a clear investigative side to the qualitative analysis to differentiate between discrete varying skillsets on the software development spectrum of problem solving and contributions. As the topic of research is based on the interest in the human metrics involved, and not simple code analysis itself, the goals set out lie within the analysis of qualitative methodologies with selective analysis to observe patterns of software design within a team.

The overall aim is to create a suite of toolsets to facilitate the mining of meaningful data to more clearly define the varying attributes of developers in a project without a pure focus on repository mining alone. Having acquired sets of metrics about mined repositories, a qualitative analysis is done to combine technical and expert systems analysis, and validate the initial results of a non-statistical exploration via the tool-set. The aims of this are to provide a solid foundation for future work within the scope of the area as a means for expansion for interesting potential correlations between code metrics.

Objectives

The ultimate objective of this research project is to determine patterns of qualities residing in software development, and generalise systematically across large datasets obtained from real-life repository analysis. As this is an extremely general set of requirements, the following is a terser set of guidelines to follow:

- The infrastructure must be initially be built as a basis for data-set generation
- The infrastructure should be as modular as possible
- The infrastructure should generate metrics per repository on a per commit basis
- The infrastructure should visualise data generated meaningfully on graphs
- The infrastructure should show clear trends about the state of a repository
- The infrastructure should relate these data to human metrics of contribution quality
- The infrastructure should allow a user to submit a repository for digestion

Report Outline

This report consists of several chapters as outlined below that form the arguments set out within this final year project dissertation:

State of the Art

Chapter 2 introduces the background of the problem space. It sets out the current state of research within the area of code analysis metrics and the expansion of their use on a wider scope.

Design

Chapter 3 sets out in detail the design and thought process for the methodology and design of the system on a global scope. Initially, the chapter starts out explaining the context of the overall system, and then proceeds to go into an experimentally focused explanation of the design of the sufficient components for the tool chain infrastructure as expanded upon in chapter 4.

Implementation

Chapter 4 sets out in a greater scope what is meant by the digestion pipeline and representation of huge amounts of data, and the operational details that follow in generating details about the operation. Usages and relationships of the tool chain are discussed in practical analysis, and then diverges into the area of the inferences for performance characteristics of these components.

Evaluation and Future Work

The final chapters, chapters 5 and 6, present inferences and conclusions pertaining to the project, dealing in-depth with areas of interest to take away from socio-metrics within the field of software engineering, and considerations for future work relating to this project.

Challenges

Throughout the project there was a constant challenge of a severe lack of infrastructure for generating appropriate metrics and analysing repositories to fully satisfy the needs of this project. Due of this lack of tool-chains being readily available, there was the additional challenge of creating an infrastructure working together in tandem to attain an automated system of Git repository digestion and analysis. In addition to this, there was the challenge of generating data sets using the toolchain, and having to explore for possible correlations in metrics without a known outcome beforehand.

STATE OF THE ART

This chapter is the basis literature pertaining to this branch of software engineering data science for qualifying and quantifying metrics associated with code analysis. Main areas of interest are set out as per Halstead, McCabe, and the concept of a maintainability index; which are contextually shown later in chapters 3 and 4 as per the tool-chain developed for code analysis. The aim of this chapter is to define and show the engineering challenges considered for devising and developing a tool-set for mining and analysing data-sets of arbitrary software repositories.

BACKGROUND

Metrics within software engineering come as part of the analysis and dimensionality of the structure and size of a given project scope to a known domain, defined formally as "a qualitative measure of the degree to which a system, component, or process possesses a given variable.". (IEEE, 1990)

Metrics give a set of classifications to aide in differentiating qualities of code repositories from one another. While the overall aggregate of software lines of code (SLOC) is a general measure of the size of a program excluding comment lines, it gives no overall classification of the designated quality of the code being implemented, whether it's overly complex, or given qualities of redundancy being contributed. (Nguyen, et al., 2007) Given this inherent ignoring of the structure of the program, this gives rise to other more in-depth metrics for ascertaining more information about the relative functionality as a unit of measurement for classification.

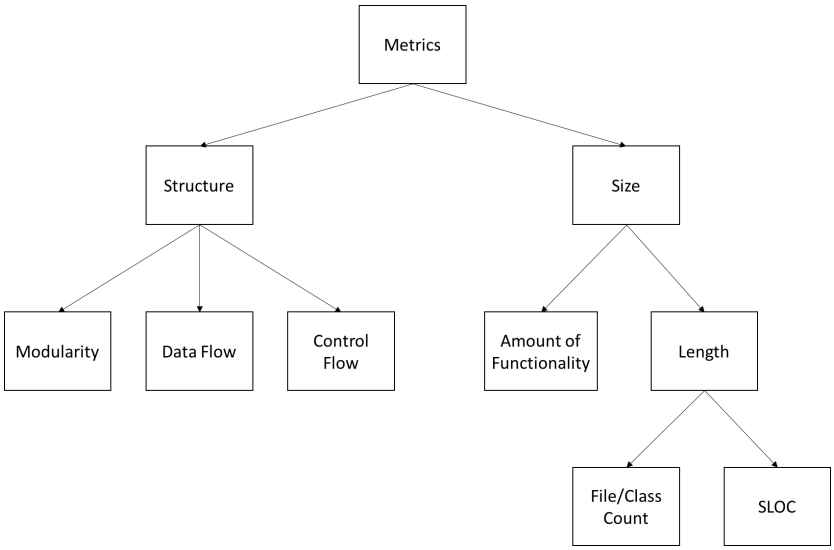


FIGURE 1 AGGREGATION OF METRICS ACROSS DOMAINS (SEREBRENIK, 2011)

HALSTEAD METRICS

Halstead metrics are an approach to make empirical and educated guesses about the current state of a program by inferring certain software metrics about it under the domain of the amount or degree of functionality. (Halstead, 1977) The measures' goal is always to remain language agnostic, and refer specifically to properties and relations attainable within software. Halstead metric analysis results in a set of metrics that not only relate to the overall complexity of a program (as per section 2.5 McCabe

complexity), but also to a set of properties with some measurable relations among them for inferring further suppositions.

Stemming from research done in 1977, Halstead metrics aim to give a further, more in-depth insight into counting the amount of unique and used operators and operands within a program. The overall aim of Halstead metrics is to infer more information rationally about a program over the easily calculable software lines of code metric (SLOC) – ascertainable simply by counting the lines of code in a program, but lacks any real insight. (Halstead, 1977) The Halstead metrics domain has several properties derived from a measurable index of code declarations within given scopes. These properties exist within four groups: the distinct number of operators (η_1), distinct number of operands (η_2), overall number of operators as an instance (N_1), and overall number of operands as an instance (N_2).

A distinct operator defines the mapping of use for an attribute such that an endomorphism is defined as that of the defining feature of an attribute. This can be the data type of a strongly typed language, the name of a function, encapsulating symbols for specific use such as brackets; corresponding to verbs, syntax, and elements that are not data.

A distinct operand is the attribute which receives interactions by operands – resulting in their use in functions and being manipulated for data storage and representation accordingly.

The overall number of instances for both operands and operators is calculated by summating the total amount of each. This corresponds to being a variable, constant, literal or expression. Take the following example C program for computing the average of four inputs:

```c
int main(void) {
    int a, b, c, d, average;
    scanf("%d %d %d %d", &a, &b, &c, &d);
    average = (a + b + c + d) / 4;
    printf("average = %d", average);
    return 0;
}
```

The output of tokenizing, given the individual operands and operators within the above program, would result in the following outputs. The distinct operators in this segment would be comprised of such, given that the function definition and the encapsulating braces for it are ignored:

int	scanf	;	&	=	+	/	average	printf	return

The distinct operands in this segment would then be:

a	b	c	d	average	“%d %d %d %d”	4	“average = %d”	0

Derived Halstead Metrics

Halstead metrics build on the base idea of operands and operators thereby giving a deeper, more meaningful estimate of properties associated with software development processes that are directly consistent with effort required with source code over 7 derived metrics (VirtualMachinery, n.d.):

Vocabulary (n): $\eta_1 + \eta_2$

Vocabulary is defined as the overall summation of individual operands and individual operators. The resultant, n, is an integer that corresponds heavily with the given amount of lines of code within the program. It gives a greater insight into the minimum amount of understanding needed to understand the program in full. (VirtualMachinery, n.d.)

Program Size (N): $N_1 + N_2$

Program size is derived from the total number of overall operands and the total number of overall operators summated together. This property gives an overall sense of the length of the program being analysed. (VirtualMachinery, n.d.)

Program Volume (V): $N * \log_2 n$

The volume of a program is generally taken to be the absolute size of the program. A general rule of thumb for estimating the correct volume is that within an observed function should encompass the scope of $20 \leq V \leq 1000$, while the volume of an observed file should encompass a larger range of $100 \leq V \leq 8000$ due to the increased amount of functions residing within a file. (Serebrenik, 2011)

Program Difficulty (D): $\frac{\eta_1 * N_2}{2 * \eta_2}$

Difficulty is a representation of the operators' and operands' given uniqueness and total usages together. (Serebrenik, 2011)

Effort Required (E): $V * D$

Effort is a derived estimation at the amount of difficulty observed with interpreting the implementation of the code within a program. (Serebrenik, 2011)

Bugs/Errors Expected (B): $\frac{V}{3000}$

Errors are derived as an estimation to the number of bugs residing in a program proportionally to effort put into the implementation of code. (Serebrenik, 2011)

Testing Time (T): $\frac{E}{k}$

Testing time takes an arbitrary variable k estimated to be a Stroud number, generally having the value of k=18. This number is dependent upon the context of development team, considering dependencies such as the background of the team, criteria of criticality level, team competency, and so on. The value of this rises and falls depending on these factors, and so is estimated with some estimation and experience. (VerifySoft, n.d.) The testing time is an estimation in seconds of the time taken necessary to write the program proportionally to effort as estimated from other a priori influencing factors.

At its core, Halstead metrics given the above six metrics, as building blocks to infer more information about a software project and its contributors. A main difference observed between SLOC and Halstead complexity, is that SLOC doesn't take unique definitions into account that are used in the program's execution – omitting any variable and definitions that remain out of the scope of use.

The following is a code excerpt as per (Serebrenik, 2011):

```
void sort(int *a, int n) {
  int i, j, t;
  if (n<2)
    return;
  for (i=0; i<n-1; i++) {
    for (j=i+1; j<n; j++) {
      if (a[i]>a[j]) {
        t = a[i];
        a[i] = a[j];
        a[j] = t;
      }
    }
  }
}
```

With some Halstead analysis on the above code segment, a more insightful set of metrics may be generated. Operators and operands are counted like the following:

Operators:

3	<		5	=		1	>
1	-		2	,		9	;
4	()		1	return		6	[]
3	{}		1	+		1	++
2	for		2	if		1	int

Operands:

1	0		2	1		1	2
6	a		8	i		7	j
3	n		3	t			

Resulting properties:

	Total	Unique
Operators	$N_1 = 50$	$n_1 = 17$
Operands	$N_2 = 30$	$n_2 = 7$

This results in the given volume of this procedure to be $V = 80 \log_2(24) \approx 392$, which in turn lies within the scope of $20 \leq V \leq 1000$; satisfying the requirements for program length. (VerifySoft, n.d.)

Values of the other derived metrics result as the following:

Expected source code difficulty (D): $\frac{\eta_1 * N_2}{2 * \eta_2} \approx 36$

Expected effort (E): $V * D = 14112$

Estimated time to understand and implement (T): $\frac{E}{18} = 784s \approx 13$ mins

Expected bugs in implementation (B): $\frac{V}{3000} \approx 4.7$

CODE CHURN

Code churn is the analysis of how source files change over time through observing what lines of code are being added, removed, or modified. It is a net measure of the volatility of a code base – based upon how frequently code is modified over time with code ownership. Code churn can have negative connotations attached to its analysis. A volatile code base represents an unclear vision of what the future of the project is for developers working on it. Indecisiveness in a code base can be a representation of unclear specifications or waste occurring in time spent implementing certain features. (Thompson, 2016)

Volatility also takes place when code review is occurring. This can be because of discrepancies in coding standards imposed by engineers on the code base stemming from implemented features that are undergoing clean up, or unfinished features undergoing a review process. The net outcome is dependent on the definition of what the team expresses “done” to be. Code can naturally vary in its quality, with higher complexity code having a lower readability value; and segmented, more generified code having a lower complexity, as explained in greater detail in section 2.5 on McCabe complexity.

As implementing new features can impose a high churn rate in pioneering additional functionality, a potential solution for mitigating indecision can be to prototype the feature through multiple paths in an exploratory fashion. Of course, churn is not always due to unimplemented features, but can also be a manifestation of the current mind set of the engineer. Churn is an inverse representation of the throughput of the engineer working on tasks. Psychologically, if the engineer is experiencing under-engagement, it can be a deeper representation of burn-out or disengagement. (Thompson, 2016)

Abstract Graph Data Type

A program's structure of execution, control flow, and data flow are generally mapped to a flow graph data type in the form of nodes and directive vertices between edges. Following this, mapping a program as such results in graph-based metric expressions to follow.

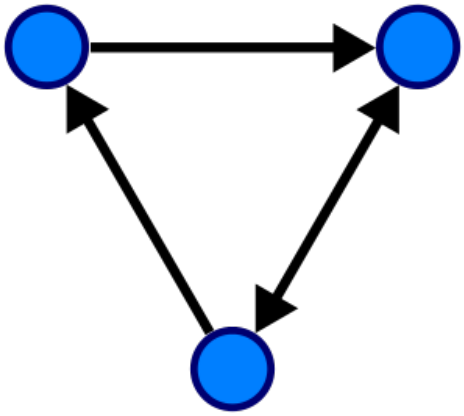


Figure 2 A 3-node directed-graph (Booyabazooka, 2006)

A graph is an abstract data type where there is a given start point, endpoint, a number of nodes, a number of vertices, and a given maximum depth for the program to be traversed from start to finish of its execution. (Goodrich & Tamassia, 2001) The control flow graph deduced as per the data structure is representative of the control flow path deduced by testing conditions within the program's run. Operations may not be immediately sequential, resulting in more complex structures of orders of execution. As each operation is reduced to a node within the graph, the connections between nodes can be thought of as the rate of complexity of flow control. There may be several possible branches and varying paths for traversing over the logic of the program, resulting in an overall differing set of complexities depending on the paths and pre-conditions chosen. This is known as the "cyclomatic complexity" of a program's execution – with individual testing metrics associated per path.

The general testing metric for the volume of a graph is given by $V(G) = \#edges - \#vertices + 2$. A control flow graph, where decisions impact on the path chosen, is given by $V(G) = \#binaryChoices + 1$, or more segmented as $V(G) = \#ifConditions + \#loops + 1$ where the given boundaries for graphs are $V(func) \leq 15$ and $V(file) \leq 100$. (Cormen, et al., 2001)

```
void sort(int *a, int n) {
  int i, j, t;
  if (n<2)
    return;
  for (i=0; i<n-1; i++) {
    for (j=i+1; j<n; j++) {
      if (a[i]>a[j]) {
        t = a[i];
        a[i] = a[j];
        a[j] = t;
      }
    }
  }
}
```

The above function can be broken down structurally to determine more than just a Halstead tokenization of operands and operators. By simply counting the quantity of if conditions with respect to their quantity of pre-conditions (2) and loops (2) within the fragment, it results in an overall volume testing metric of 5; which is within the boundary for structural complexity of 15 for a function.

McCabe Complexity

McCabe cyclomatic complexity is a quantitative metric that aims to reduce metrics of code complexity down to a number by analysing linearly independent paths of complexity throughout the source code with a flow direction affixed. Using a graph data structure, it follows that the source code's flow of execution to its most basic logical inference of path deduction is reduced. In sequentially ordered operations, there is a flow loop throughout the program's execution that can be represented by nodes with a connective direction between them. (McCabe, 1983)

The magnitude of complexity is linearly dependent on the amount of conditions imposed on its execution. For example, nesting conditional loops, or having loops with multiple conditions for execution dependence increases the order of magnitude for complexity of the graph. For instance, a program with a single loop with one pre-condition would have the resulting cyclomatic complexity of 1; whereas nesting another inside this with two pre-conditions would result in a complexity of 3. This is brought about by logical deduction that a Boolean expression can have the outcome of being either true or false. As these statements can be evaluated as only one or the other, the series of pre-conditions for their execution can take several paths for choice based upon the steps leading to there.

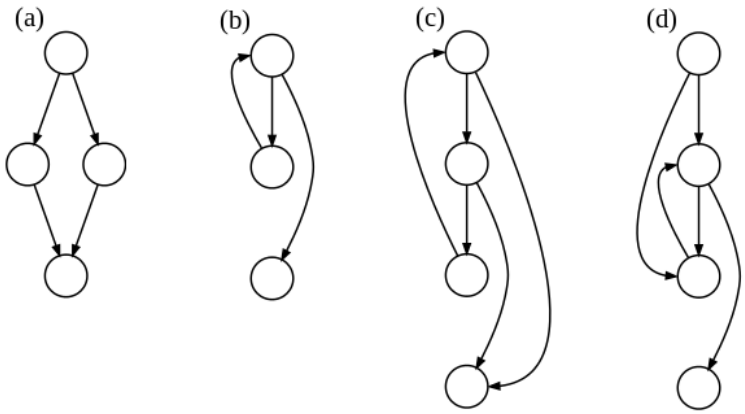


Figure 3 (a) if-then-else loop (b) while loop (c) while with sentinel (d) for loop (Dissanayake, 2014)

As can be seen in all flow graphs, there are two common points – an entry point and an exit point. (Harrison, 1984) Logically, the more paths that can be traversed in a program, the more complex a program can be, as it can extend through multiple choice permutations. Mathematically, the cyclomatic complexity can be described as a program broken down into its most basic blocks, and the relationship between them as a directed graph as connective edges inferring direction of movement. As the complexity is therefore dependent on the number of nodes (N), the number of edges for nodes (E), and the number of connected components (P); the resulting complexity (M) is as follows: (Harrison, 1984)

$$M = E - N + 2P$$

Depth-First Search

Depth-First Search, or DFS, is a traversal method for the abstract graph data type determining the graph's exploration route from start to end over a given vertex, and a topological adjacency list for vertices with respect to the given node's neighbours. The search algorithm is of O(N) complexity, traversing paths until an impasse is reached, causing the algorithm to backtrack on its path to reach other unexplored paths of its neighbours recursively. DFS uses the graph classification of edges for determining the most appropriate set of paths to traverse to generate routes from subtree to subtree and generate the best route from top to bottom over a vertex recursively. (Goodrich & Tamassia, 2001)

```
dfs(V, adjacent)
  parent = {}
  for s in V:
    if s not in parent:
      parent [s] = null
      dfs_visit (V, adjacent, s)


dfs_visit(V, adjacent, s):
  for v in adjacent[s]:
    if v not in parent:
      parent[v] = s
      dfs_visit(V, adjacent, v)
```

This then leads to the idea of reachability within the closed system as a method for optimisation of source code. As reachability is only possible if code execution extends to that block, unreachable code can be safely removed without any unwanted effects as these blocks do not contribute anything to the program. The inverse may also be true where a program may be stuck in an infinite loop of execution without any way to break out of the iteration, and isolate the execution to only a closed sub-system in the graph traversal. (Even, 2011) Graph structures generally form trees that are traversed in order by DFS by starting from the root node, and generally explore as far as possible along certain routes before coming to an end. The aim of DFS is to maximise the traversal depth before having to backtrack to another branch.

Maintainability Index

The maintainability index is a metric tying together the three metrics: Halstead volume, McCabe cyclomatic complexity, and the raw number of lines of code in one in-depth measure of the complexity to determine the relative ease of maintaining a fragment of code. The boundary conditions for quality lies over a discrete set of four distinct areas: less than 0 is grade D, 0-65 is grade C, 65-85 is grade B, and greater than 85 is grade A. (Naboulsi, 2011)

$$MI = 171 - 5.2\ln(\mathrm{V}) - 0.23\mathrm{V}(\mathrm{G}) - 16.2\ln(\mathrm{LOC})$$ (Radon, n.d.)

There is another implementation of maintainability index, considering a separate metric of the quantity of comments within a program's scope and definition. This builds on its base definition of tying together multiple contextual and meaningful metrics. As code alone can only be interpreted, comments fast-forward this process by defining a use and an expected behaviour of a subroutine. However, this can only be true if the comments provided are meaningful and provide a useful insight, which is again a continuous and potentially indeterminate metric that varies among individuals.

$$MI_0 = MI + 50 \sin \sqrt{2.46 * CM\%}$$ (Radon, n.d.)

Working again with the same sorting code sample, the maintainability index will be calculated with respect to other previously acquired metrics.

```
void sort(int *a, int n) {
  int i, j, t;
  if (n<2)
    return;
  for (i=0; i<n-1; i++) {
    for (j=i+1; j<n; j++) {
      if (a[i]>a[j]) {
        t = a[i];
        a[i] = a[j];
        a[j] = t;
      }
    }
  }
}
```

Given the Halstead volume (392), McCabe complexity (5), and given SLOC excluding the function definition (14); the maintainability index governed by this implementation resides at approximately 96. This value is within the area of being greater than 85, and so is of grade A – easy maintainability.

Modularity

As functions and objects within a program aren't always independent, there exists sets of coupling and cohesion in tying together functionality and modularity. A cohesive module refers to a one that makes many intra-module calls within its own scope, showing a low dependence on functionality outside of it. The coupling of a module refers to the inter-module calls that a function must make to complete a subroutine. (Serebrenik, 2011)

In addition to these clashing features of program structure, there exists two modularity metrics known a fan-in and fan-out. The fan-in index of module M is the quantity of modules making calls for a subroutine within M. The fan-out index of module M is the quantity of module calls made by a function in M. These indices are important as it shows the levels of code use and reuse within a program. A module with a fan-in index of 0 implies that there is a level of redundant or dead code existing within the system that isn't in use by any module. (Borysowich, 2007)

By defining the fan-in (read) and fan-out (write) indices for a given module's functions, the information flow can be determined over a global data structure of connecting inputs to outputs. The generalisation of connectivity of modules within a system results the following definition: (Serebrenik, 2011)

Shepperd index (s): $(\text{fan-in} * \text{fan-out})^2$

Henry & Kafura (HK): $\text{SLOC} * s^2$

DESIGN

This chapter set out the details of the design implemented using the literature of the previous chapter as the basis for its methodologies in choices taken. It sets out explaining details across major sections of system topology, technologies used in detail, design patterns implemented, as well as security considerations for the wide-scale use of version control systems. Given the prevalent use of JSON across the web through RESTful services, detail is also given to the storage methods and data abstractions deployed in data generation and protocol deployment.

BOTTOM-UP METHODOLOGY

Experimental design played a large part in undertaking this project. For exploration to be done on tackling the distinct problem of relating code to human metrics, there was a lack of tools available for leverage. Dealing with this issue led to the design and implementation of a toolset for a specific pattern of use for this specific problem – in essence, building a suite of highly tailored tools to fit the problem at hand.

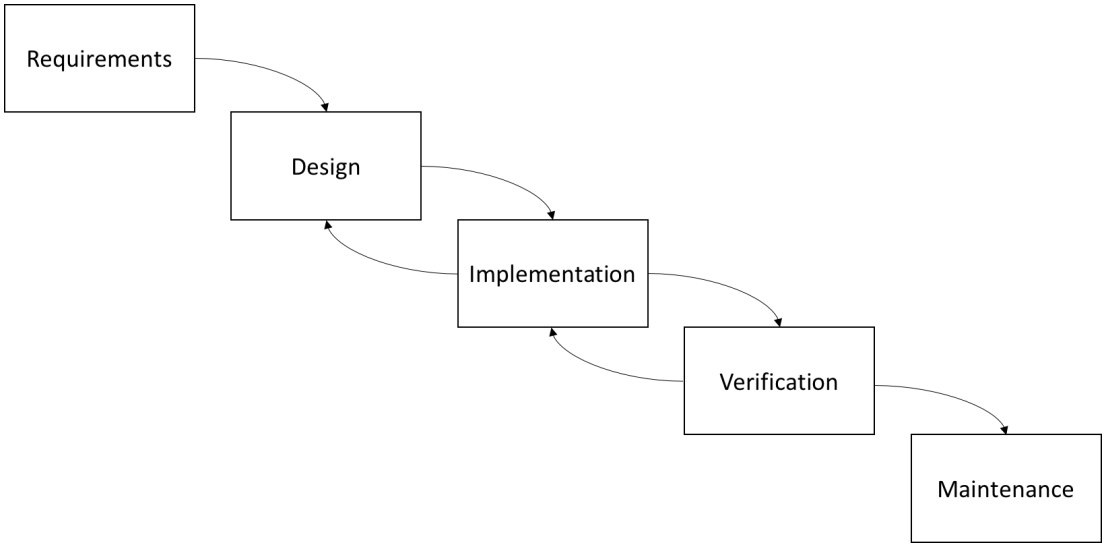


FIGURE 4 DESIGN PROCESS GENERALISATION

The methodology throughout the development and implementation has always been to work towards the end goal by starting small and iterating often as per Agile methodology design principles. The design of the system follows a bottom-up design – initially with small interactions of having a Python script successfully clone repositories and collecting meta data, to that of a fully-fledged pipeline system with a feed-forward design from stage to stage. This consortium of modules working independently for one single job led to a very modular design, allowing for a variety of simple sub-systems to work together cohesively and coherently in tandem with the desired outcome of data visualisation and metric aggregation.

Another item for consideration was the premise on which the project was started. The School of Computer Science and Statistics in the University of Dublin, Trinity College, is currently carrying out research on the research topic of systematic code analysis. As a by-product of working with Professor Stephen Barrett, there was an exposure to the potential of the system being used in further work as part of a module in the existing system. Ergo, the platform implementation should have the potential

to be agile enough to work as a module in an already existing system. As an increase in discrete encapsulation results in a higher amount of modularity, it was decided that it was of utmost importance to allow for the system to be as modular as possible to facilitate the potential use of this work in time to come for others. Bottom-up design allowed for the project's various areas of the system to be completely encapsulated in their own modules. The independence between modules meant that each individual part of the system could be developed in isolation, ensuring that changes made created minimum dependency injections and avoided God-object refactoring code smells. (Fontana, et al., 2011)

System Topology

The implementation is composed of several nodes within a pipeline working in tandem together to accomplish various tasks from starting a job to visualising it in a graph format. The user interacts with a web interface to delegate and access jobs, where a URI to a GitHub repository can be submitted as a job through an asynchronous post request through Ajax to the backend. The endpoint on the Node.js backend dispatches a job asynchronously by invoking Python scripts through the shell and passing the parameters necessary for Git and jobs to complete successfully.

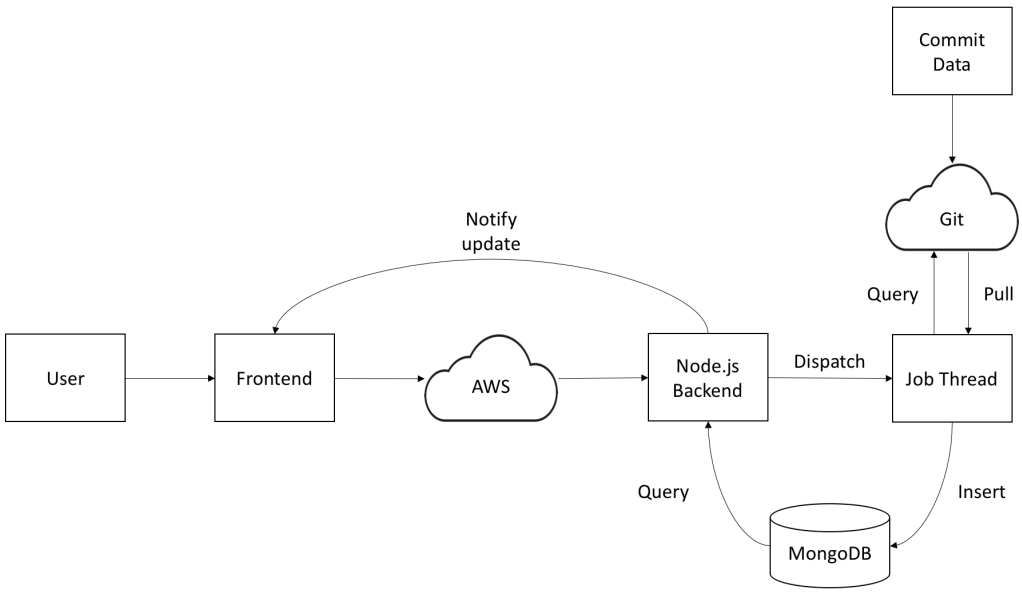


Figure 5 System topology

The Python job initially clones the repository and pulls necessary commit data by paginating through the GitHub API. The necessary commit data is stored in a list data structure and is then iterated through, at each stage resetting to the appropriate commit SHA head, and running metric analysis until the end is reached. The Python job constantly writes to a MongoDB backend, while Node.js reads constantly and notifies the front end of any changes in progress using the emitter design pattern via the MVC architecture (as discussed in section 3.5 in Design Patterns). As the jobs progress, the data visualisations on graphs can be viewed under appropriate tabs. There is a variety of graphs that aims to link code to human metrics pertaining to complexity, contributor join time, and SLOC; as well as derived metrics with these as the base form of generation.

Git Version Control

A version control system is a centralised way to keep a project in check and to collaborate with other developers. Git is a tool that follows this requirement by being a version control system for collaboration work on projects using a distributed architecture over a tree structure. Due to its distributed nature, it follows a non-linear version history. (Atlassian, n.d.) As there is no centralised repository for delegating control to a central authority, it's paramount there be a way to commit and merge differing histories together and mesh together to form a consistent schema. To combat this divergent set of histories, it uses a SHA-1 hash to keep repositories in line with commit changes and ordering for verification. This has the by-product of being able to encode an entire repository history towards one commit in the hash. (Sink, n.d.) Older centralised control systems such as SVN and CVS are required to synchronise with a central authority first before commits can be submitted to the repository. Git was chosen as it is the current standard for the software industry version control.

One of the primary things needed for analysing a repository is to be able to utilise Git programmatically to access a centralised repository store. It was deemed necessary to implement a simple system for command line interaction programmatically, where a repository can be cloned and acted upon easily. Once the repository is cloned to the local folder, its history can be iterated over, and GitHub's API can be leveraged to record more meta data about the change history. GitHub is an extremely popular website for developers to store projects and collaborate with other members easily. There is also the advantage of not only having an exorbitant number of active repositories, but an extensive API for easy interaction with projects. This API allows for scripts to be authenticated and access subsets of data across repositories as appropriate to the level of details. Through use of this set of web services, it's possible to use the API to generate a list of SHA keys per commit, sanitise the results, and then use these values to iterate over the commit history from start to the most recently committed change.

Language Choices

For the main processing jobs of the project, Python was chosen as the primary language. Initially, other forms of programming paradigms and languages were weighed up; for instance, using a functional programming language such as Haskell for job dispatching and dealing with the analysis of repos due to its terse nature and mathematical foundations. However, Python allowed for accomplishing more in a shorter amount of time while still maintaining high levels of readability. Python was used as the primary scripting language due to its pseudocode-like nature. The Python programming language is a level of abstraction above C, and is the de-facto lingua-franca of data science and is a general-purpose programming language highly suited to rapid-prototyping, namely due to Python's aphorism of terse, pragmatic code with a modular nature.

For the bulk of processing for data representation and job dispatching on the server-side, I decided that JavaScript would be well suited for uniting an interface under one language. As a multi-functional and multi-paradigm tool that can work on both the client and server stacks, it was found that JavaScript was well suited for the nature of the task at hand. The design of the system incorporates a dynamic frontend and backend. This results in there being two main areas of use for JavaScript throughout this stack – Node.js on the backend for handling endpoints and aggregating database querying, and jQuery on the frontend for dynamically setting content and creating requests asynchronously through Ajax. The asynchronous nature of JavaScript also resulted in a high level of performance for regular accessing of repo metrics for graphing.

Design Patterns

The model-view-controller architecture is a common pattern for designing the layout of interactions among modules in software design today. Its main argument for use is the extraction of dependencies from code, and increasing the level of modularity in a code base. It consists of three distinct areas – the model, the view, and the controller which work in tandem with each other to separate the logic from the data and view controllers.

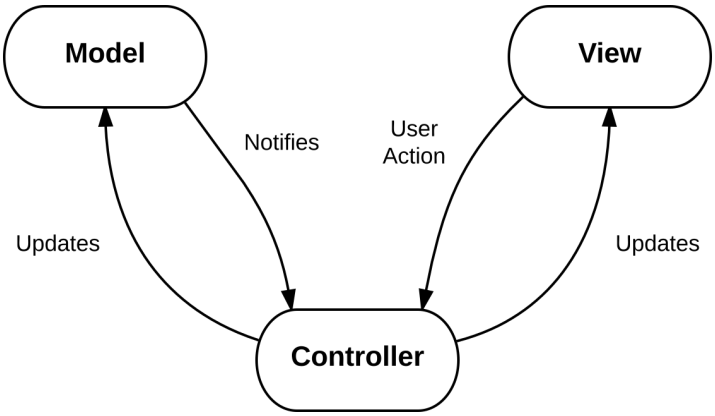


Figure 6 MVC architecture illustration (Google, 2016)

The model portion of the model deals primarily with handling the behaviour of the application by responding to certain events and notifying any observers which then trigger events to happen with a persistence layer of data structures or a database. The view element of this architecture is the interactive user interface that works to provide a way for the individual to effectively see that rendered data to the screen. The controller acts as a middle-man between the user interface view, and the persistence layer. The controller delegates actions and works to receive actions and calls, which then correspond to performing given subroutines. (Reenskaug & Coplien, 2009)

The emitter-observer design pattern works as part of the MVC architecture by creating a list of observers and dependent elements, and notifying accordingly with respect to updates in state changes. (Celis, 2010) An emitter generates an event-driven change according to a state change of an object within the system. An observer watches an emitter for any changes notified during its existence, and reacts accordingly by triggering other actions on some functionality occurring.

Dependency injection is a paradigm of providing an inversion of control to classes, where the objects for a method signature are provided through parameters as a dependency instead of instantiation within the method itself. The inversion of control of dependency construction is quite useful in testing phases within development cycles; as it allows for objects and data structures to be mocked. The outside source of providing dependencies gives a greater amount of control to run-time object injection.

Programmatic Analysis

As each commit within a repository can be iterated over and the commit's meta-data be retrieved from GitHub's public API; it's possible to generate more metrics about the actual code changes that occurred within that commit. While there are some tools available for broad forms of code analysis via various sources, to find a tool that appropriate covers the necessary metrics required was a challenge. The base requirements for finding such a tool revolved around finding a large set of metrics about lines of code, relative complexity, and measures about the code's intrinsic maintainability. Tools around code analysis generally exist within the domain of ensuring that code remains to a certain standard, notifying if any violations of the current enacted standard have occurred by members of the team, as well as ensuring that certain base design principles and patterns are to be followed.

SLOC/LOC, as discussed in previous chapter, is the most basic measure of a program. It is the count of individual lines of code in a program that is generally tied to the effort of development. SLOC is considered a poor measurement due to the lack of standards for counting lines of code. It is generated in analysis due to the part it plays in creating maintainability metrics. As per section 2, Halstead volume is a general measure for empirically measuring the absolute size of a program with respect to the overall vocabulary of a program in terms of operators and operands. In addition to generating the derived McCabe complexity with a program's SLOC, the maintainability index of a program can be defined which corresponds to the general ease of maintaining a program. The maintainability of a program gives a deeper sense of meaning to the overall intrinsic difficulty of the algorithm being implemented.


```
{
    "_id": "58e62a5e9bb1101acc8cea01",
    "AI-Art/aiart/KMeans": {
        "mi": 88.24222840912691,
        "rank": "A"
    },
    "commit": "c7e0b0f",
    "commit_details": [
        {
            "_id": "58e62a5d9bb1101acc8ce9ee",
            "sha": "c7e0b0f59a6cc718875945220fccda288861ad45",
            "head": "c7e0b0f",
            "author": "Z2F2a3IuaXJlQGdtYWlsLmNvbQ==",
            "time": "2016-08-08T09:31:09Z",
            "iteration": 1,
            "max_iterations": 8,
            "repo": "AI-Art"
        }
    ]
},
```


Figure 7 Sample maintainability index data for commit c7e0b0f

Analysis and iteration of commits generate large volumes of data per repository. The high volumes of data coupled with a relatively relaxed schema leads to the use of a document-oriented backend for storage. Initially, volumes of analysis metrics were stored as a simple JSON document. As this quickly became unfeasible due to huge quantities of generated data, the next stage of design was to incorporate a backend into the system with some form of persistence layer for quick and efficient access.

Open Source Software

Open source software refers to publicly available repositories that can be modified and shared freely due to the design of being openly accessible by the internet on sites such as GitHub. This paradigm of software development allows for the enhancement of the source code by mass collaboration efforts of regular users. Open source software is becoming an increasingly more popular method for developing tools for the masses; with resources such as the Linux operating system and Apache web server having already been chief pioneers into the exploration of this paradigm resulting in their mass adoption. (Paul, 2006) There are many open source software repositories with large quantities of commits and a large user base of past and active developers providing new features, refactoring, and committing bug fixes regularly. This ecosystem provides a large amount of tangible data that can be easily mined and aggregated accordingly to metrics derived from human interaction over commits for a repository.

Documented-Oriented Storage

Document oriented databases are very powerful tools in horizontally scaling system architectures where large volumes of data are common. NoSQL is a document-oriented database that allows for documents to be stored persistently in JSON format using key-value stores in an opaque manner. NoSQL pertains to the need of storing unstructured data in a logical and efficient manner – namely for the storage of documents that may not have a set schema structure. It also allows for an evolving document schema over time, where a set schema doesn't have to be defined initially, and can remain completely unstructured in document insertions. As Amazon Web Services allows for multiple EC2 instances to be clustered together and work horizontally as a system stack (as per Distributed Cloud Computing in section 3.9), it clearly makes sense to use an appropriate persistence layer NoSQL architecture.

JSON is well suited to representing the large quantities of data within the scope of this project due to the lightweight design architecture. It was chosen to resolve the task of data aggregation and the delivery of data to a service's endpoint due to the natural pairing with JavaScript as per its object notation. JSON builds on a key-value pair, being realised as a natural collection in various languages through hashing a value to a known key name through serialisation. (ECMA, 2013) This standardised method of holding data unifies the front-end scripts with the back-end and persistence layers with one given method of purposely representing data meaningfully.

As the size of data sets grew over time, passing bulk amounts of generated data over thousands of commits to the front end became unfeasibly slow or resulted in broken data aggregation. The design of program flow was then altered to allow for promise-based callbacks. Promises allowed for data to be asynchronously requested and passed synchronously to further chains of dependencies. Server-side generation was used to groom the data on a per-commit basis, allowing for fata to arrive pre-groomed and ready for iteration on graphs.

```
{
    "_id": "58e228ea9bb110c085ea43a8",
    "files": [
        {
            "loc": 16,
            "lloc": 16,
            "single_comments": 0,
            "sloc": 16,
            "multi": 0,
            "comments": 0,
            "blank": 4,
            "file": "AI-Art/aiart/DataGenerator.py"
        },
        { … }, // 8 items
        { … }, // 8 items
        { … }, // 8 items
        { … }, // 8 items
        { … } // 8 items
    ],
    "commit": "c7e0b0f",
    "average_complexity": [
        {
            "_id": "58e228e89bb110c085ea439a",
            "commit_head": "c7e0b0f",
            "avg_complexity": "1.46153846154"
        }
    ],
    "commit_details": [
        {
            "_id": "58e228e89bb110c085ea4399",
            "sha": "c7e0b0f59a6cc718875945220fccda288861ad45",
            "head": "c7e0b0f",
            "author": "Z2F2a3IuaXJlQGdtYWlsLmNvbQ==",
            "time": "2016-08-08T09:31:09Z",
            "iteration": 1,
            "max_iterations": 8,
            "repo": "AI-Art"
        }
    ]
},
```


FIGURE 8 SAMPLE RAW LOC DATA FOR COMMIT C7E0B0F


```
{
    "_id": "58e62a5d9bb1101acc8ce9ee",
    "sha": "c7e0b0f59a6cc718875945220fccda288861ad45",
    "head": "c7e0b0f",
    "author": "Z2F2a3IuaXJlQGdtYWlsLmNvbQ==",
    "time": "2016-08-08T09:31:09Z",
    "iteration": 1,
    "max_iterations": 8,
    "repo": "AI-Art",
    "loc": 137,
    "average_complexity": "1.46153846154"
},
```


FIGURE 9 REFACTORED SERVER-SIDE GENERATION TO LOC AGGREGATION TO COMMIT C7E0B0F

Distributed Cloud Computing

The client-server model refers to a network design architecture, where there is one centralised host and many clients connected to it. The centralisation of control allows for clients to connect to the server, and then request resources or services for delivery of data back to the client. This form or architecture is one of the primary methods which the internet is built upon – where the cooperation across machines results in the overall delivery of something that has been requested.

Amazon Web Services offers a plethora of services for distributed and clustered cloud computing on-demand as an infrastructure service. Due to the nature of high volumes of data processing being done locally on the machine, it was decided that it was much more feasible to use an EC2 instance and scale it accordingly to the work being done by the worker threads for jobs requested for harvesting. Scaling allows for more instances to be added to the infrastructure, leading to increased power and flexibility of this system, and ergo to much quicker concurrent operations being distributed over more virtualised resources in a data centre. (High Scalability, 2016) The increased cluster power allowed for more jobs to be done more efficiently and quickly rather than simply on a laptop. By deploying the application to an instance, it also meant that the web interface and database could all be deployed and be made accessible accordingly in a much easier fashion, thereby allowing for the project to be hosted in one central VPS location and be scaled to suit the needs of the load heuristically.

Security Considerations

Use of the GitHub API led to the first consideration for data security – credential leaking. Services like Git, as well as other canonical sources such as npm, are used to host a variety of data types for users to download, update, and make use of. Therefore, it's possible to accidentally leak passwords, secrets, keys, tokens, and SSH private keys online; giving the potential of unwanted access to individuals of these accounts if this risk is overlooked. Attackers have the potential to push remote fixes to canonical repositories given the correct credentials. Given an npm token or a GitHub deploy key, an attacker can possibly push malicious code to consumers of open-source projects; having the extremely undesirable effect of being implicitly installed en-masse. (Podjarny, 2015)

Going beyond Git and npm; the hosting of an open-source project can be accessed given the correct AWS or SSH keys being accidentally pushed. This can wreak havoc on a system – giving the attacker a range of possibilities from giving access to your system, to running up bills for computation. While this is abhorrent, it can be mitigated with the correct preparatory procedures in using Git as a tool. Avoiding blanket usages of wildcard characters in creating a commit and carefully curating the needed files to be included avoids easy captures of files that weren't intended to be shared.

Making use out of the "-p" flag when adding files allows the committer to carefully review what is being added to the staged changes. Using an ignore file allows for certain filenames, file-extensions, and folders to be always excluded from the staging area, and becomes invisible to the publishing tool. Creating exclusions for commits allows for not only accidental inclusion to be avoided in the file-system, but also removes a share of bloat within a repository by excluding build folders. However, it's important to note that configuration files, docker files, output files and certain scripts within the repository can harbour accidental key inclusion.

Other mitigations include using hooks and environmental variables for holding keys. A hook is a mechanism used in Git to perform checks before completing a commit via the command line. This has

multiple uses as a tool, including enforcing code quality checks and automated testing; but also has the additional advantage of breaking a commit in the case of accidentally including certain patterns that may be a credential. (Podjarny, 2015) Environmental variables keep credentials out of sight and hidden in the operating system, and thus avoid the problem of residing in a repository for use.

While these methods work well for data security, if credentials have already been leaked, then the only option left to do is to invalidate passwords and tokens as quickly as possible. Removing commits and resetting to a branch with the deletion of historical references do not suffice in mitigating leaking credentials. Copies of the credentials will still exist on some machine, and so cannot be trusted to be completely clear after a reset. The solution is not to change the past, but to change the future access of the key.

Another consideration within the scope of data aggregation via GitHub is the exposure of individuals' emails to the web publicly. The knock-on effect of this is that web-crawlers exist and freely roam through the internet – constantly collecting and harvesting as much data as possible from any available sources. As emails are valuable to spammers; the effect of mass-email harvesting from public GitHub repositories is that they can be sold en-masse or used directly for spamming, recruitment, or other purposes. While there is an option on GitHub to keep email addresses associated to the account private, the default value is being publicly visible as part of the Git commit object, and it's not commonly known that this is something that is freely published. (Hakes, 2016)

Given this fact, only a small section of users has activated this feature to obfuscate the associated email. While it's now a case that all future commits from an account to have a privatised email address that can't be harvested by bots; older commits before this change was made still have the old, public email address associated with them. Git does offer the ability to rewrite history and change repository histories, but this in turn is only considered for use where important data was leaked, and a mitigation attempt be enacted.


```
"sha": "97d5c554662fceda71b5b3172bf680234dee308c",
"commit": {
    "author": {
        "name": "[redacted]",
        "email": "[redacted]@cs.tcd.ie",
        "date": "2017-02-15T09:42:11Z"
    },
    "committer": {
        "name": [redacted],
        "email": "[redacted]@cs.tcd.ie",
        "date": "2017-02-15T09:42:11Z"
    },
    "message": "tweak email",
    "tree": {
        "sha": "3679e6b19f8195164e1d7c3973fd9a98febf0f2c",
        "url": "https://api.github.com/repos/sftcd/cs7053/git/trees/3679e6b19f8195164e1d7c3973fd9a98febf0f2c"
    },
    "url": "https://api.github.com/repos/sftcd/cs7053/git/commits/97d5c554662fceda71b5b3172bf680234dee308c",
    "comment_count": 0
},
```


FIGURE 10 SAMPLE COMMIT IN JSON WITH A REDACTED NAME AND EMAIL

Consequently, it's also possible to push unsigned commits to a repository and be verified as a different account. If the host of the project becomes compromised and commits have not been signed using GPG, it's possible to falsely associate commits to another account. Using GPG, this means that signing

with the hash of SHA-1 (excluding vulnerabilities within SHA-1 itself) will forever state that the entire history of the given commit for the associated tag is a trusted entity using the "-s" flag.

From analysis of various projects of different sizes; it's uncommon to see contributors with the hidden email option enabled. Knowing this, available email addresses can be harvested at the bat of an eye with a simple search of an "@" symbol. While GitHub provides a standardised web interface to identify users and push code to publicly available repositories, Git is just a means of collaboration that may or may not be publicly available to the whole world as per public-facing information. (GitHub, 2017)

Email harvesting is the use of a bots to crawl through the web to create massive lists of curated email addresses for use in advertising, phishing, and mass recruitment. The abundance of freely available email addresses to be harvested by bots crawling through the internet has led to some interesting repositories exploiting this. Back in June 2016, a certain repository was brought to the attention of GitHub that exploited the trivial use of the GitHub API to harvest emails of users between 2011 and 2014. (cirosantilli, 2016) The hoard of 5.8 million unique GitHub emails was subsequently removed.

Implementation

The previous chapter dealt with the design of the system topology regarding individual technologies associated within the tool-chain as per literature discussed in chapter 2. This chapter deals with the direct implementation of the tool-chain as a method of aggregating meaningful data sets over a large commit basis, pertaining directly to the case-by-case evaluation of metrics to directly aggregate and observe relationships between committer and code quality. The resulting inferences are then dealt with directly in chapter 5 where a conclusion is discussed having evaluated possible evidence derived from logical arguments obtained in data visualisation suite.

Operational Details

My scripts run with parameters of the account and repository requested to mine data about via the command line or through the web interface. As a job is invoked, metrics are generated on a per commit basis spanning five collections per repository database – average complexity, commit details, cyclomatic complexity, maintainability indices, and raw metrics. The script takes a while to retrieve all the commits for the repository on start of the script, due to the limitation imposed by GitHub API where the maximum pagination index retrievable is 100 records at a time. Larger repositories, such scikit-learn with over 20,000 commits; take a substantial amount of time to iterate through the initial grooming process of aggregating basic info such as commit SHA-1 head references and author data.

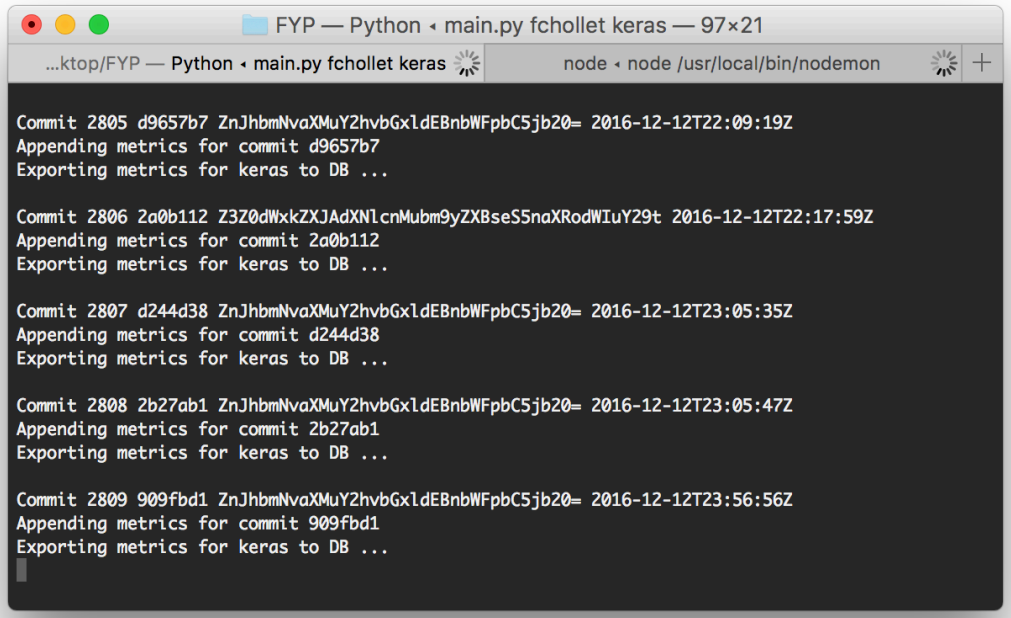


Figure 11 Progression of a Job Dispatch over a Repository (Keras)

As metrics are aggregated through the analysis process invoked, large quantities of data are generated and stored within MongoDB. Repositories with a large quantity of commits take sufficiently longer to generate and use large amounts of space for storage. This creates delays on retrieving information and

displaying it in the browser due to the large quantities of JSON data. In one instance, 6000 commits generated approximately 850MB of metrics, and taking over 18 seconds for the data to be loaded from the database into the first promise resolve of data manipulation for charts. In larger repositories, this delay can quickly become unfeasible and would require more server power and generating graphs on the server-side with the increased computational power available.

Metrics generated per commit have the owner's identity obfuscated for privacy using a 64-bit hashing algorithm on the committer's email address. This resolves issues associated with identity and relating digesting metrics and relating them back to an individual, rendering it impossible to recognise an individual's identity from its corresponding hash by simply looking at the chart.

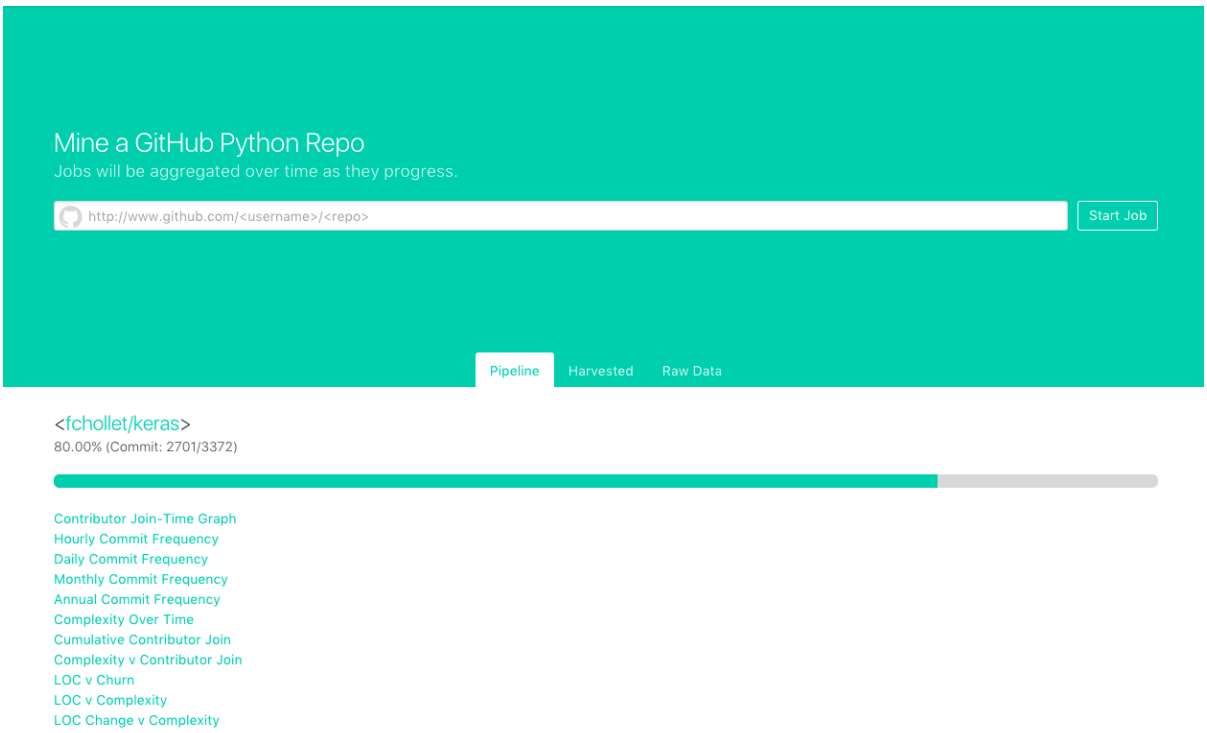


FIGURE 12 EXAMPLE JOB PROGRESSION AND OPTIONS FOR GRAPHS

Job dispatching is simplified through an easy to use front end interface, where the user must input a GitHub repository link and click on "start job" to delegate an analysis job to start. This interface allows for the user to view raw data metrics and graphs as appropriately through tabs that receive updates from the server corresponding to progress bars, showing the current progression of the repo analysis.

The use of Ajax allows for parts of the page to be updated on receiving data updates and on-click listeners asynchronously without having to refresh the page. The interface itself is hosted on an Amazon EC2 instance and runs services remotely. Its primary use is for tying together all parts of the system together into one easily accessible front-end subsystem, where the user can see the available options for data aggregation and access as appropriately.

Data Visualisation

Relationships generated across data sets raised some interesting inferences. By feeding repositories through the digestion process of analysing code metrics, clusters of variables were manually aggregated and observed; which were then graphed against one another to identify potential relationships. Metrics acquired inhabited a set of collections:

- Complexity scores
- Committer meta-data
- Raw metrics

On a per commit basis, documents were generated pertaining to a file-by-file basis associated to changes made per committer. Contextually, this gives a large scope at observing the ownership of a repository, given the pretence of the quality and quantity of the contributions resulting in a standardised complexity.

Figure 13 Sample Metric of Cyclomatic Complexity

Performance Characteristics

The performance of the system is generally slow due to the time take to iterate throughout thousands of commits per repository. On a commit-by-commit basis, the system must retrieve all meta-data about commit from the GitHub API web service, clone, then iterate through while generating metrics per given commit and insert documents appropriately into the database. The resulting outcome is a procedure that is highly dependent on the relative amount of code to iterate through in SLOC, the number of commits, and the number of collaborators.

On progression of this job, the time taken to load and view graphs also depends on the relative size of the commits and meta-data quantity associated with the repository. Graph loading time extends further and further given more and more data sets to work with, due to the issues in delivering thousands of records and generating graphs accordingly to the data set. The net result of the data delivery performance is the general increase of waiting times as the repositories age and gain volume of work. In one such instance, generating an aggregate of four repositories' Halstead complexities over a normalised time-period took approximately 33.5s to deliver to the front end and be resolved.

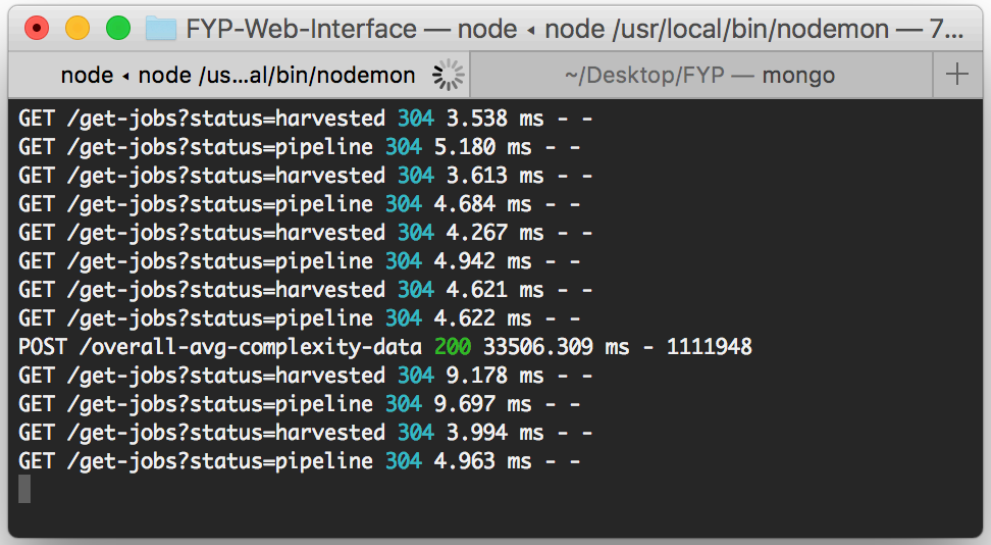


Figure 14 High Volume of Data Delivered with a High Wait Time

Join-Time v Contribution Quantity

The join time of a contributor is highly important for their overall influence of code ownership in the repository. Open-source repositories tend to undergo a snowballing effect of mass contributor adoption as the repository progresses in age. As observed from frequencies and sizes of commits in the contribution overall to the project's goal, it's frequently observed that individuals who join a repository at an initial formation stage tend to become major contributors in the project's lifetime. The effect of this is a high number of commits coupled with a large portion of code ownership in terms of the time at which an individual began contributing. As time progresses and the repository ages, a negatively exponential curve is observed where subsequent contributors, while larger in number, tend to contribute less.

Adoption Snowballing v Complexity

Leading on from the effect of join-time of individuals to a given repository, another metric inferred is the effect of the rate at which new contributors join a repository. Metrics observed across repositories show a steady increase in contributors over time, resulting in the progression of the maturity level observed in a repository as it ages. The net effect of contributors joining a repository tends to grow over time, allowing for code to become more generic over time due to many individuals dealing with the code base on a regular basis.

As GitHub is a social platform for developers to interact with one another through modifying a code base, understanding the code base's social metrics is important for determining the popularity and potential for the repo to snowball contributors over time. Repositories with an active set of members contributing at regular intervals, which also addresses an issue in a new light. Given exposure, a critical mass of users is attracted to the repository over time, which in turn places the repository in the charts for other budding developers to find and contribute to. As open-source developers receive no extrinsic motivation in pay or employment, organic growth through an intrinsic motivation becomes key to the repository's success in terms of the interest that it generates. Individuals joining the repository must generate an interest of some sort, as soliciting for collaborators will only bring a repository so far extrinsically as per the swarm size of the repo collaborators.

Stages of Repo Complexity

Repositories existing within the open-source world tend to go through continuous iterations in maturity levels over time. These stages can be discretely organised into three broad areas:

- Young
- Growing
- Mature

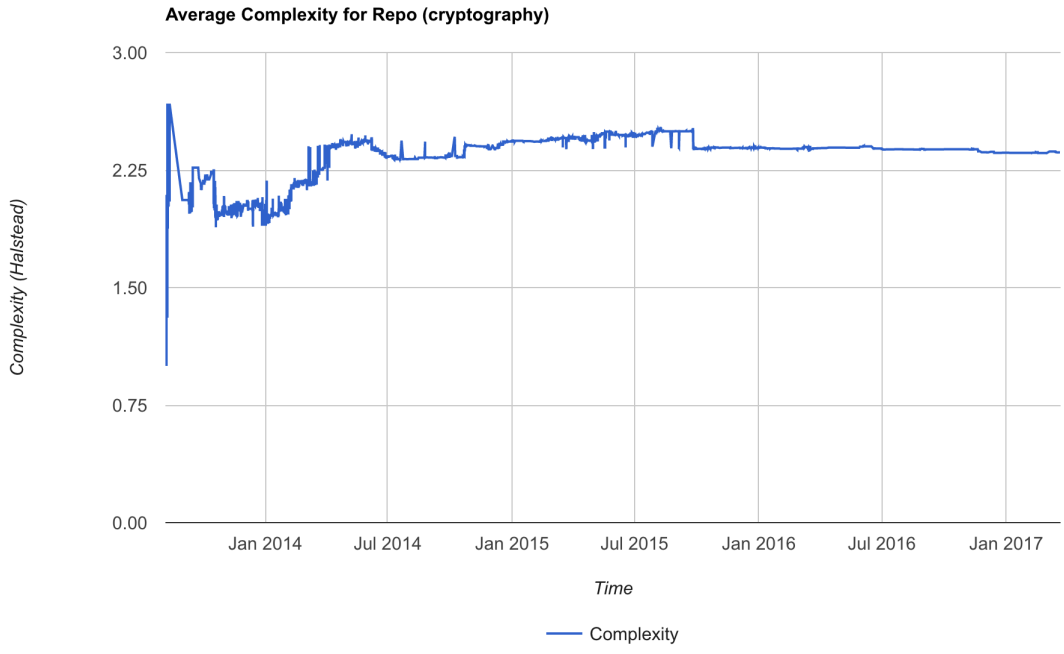


Figure 15 Repository Maturity over Time

The young stage is made up of a small group of participants initially, but tends to remain persistent throughout the project contributing the most individually as per generally having the highest amount of commits in comparison to any other later joiners. This group also shows the greatest changes in complexity for the repository, with a rapidly evolving and changing code base. The complexity within this stage rapidly fluctuates upwards and downwards.

The growing stage of software development tends to show a larger group of individuals collaborating to create software with a more fixed goal. The participants joining at this stage are higher in numbers in comparison to the original adopters, forming a larger overall group; but provide fewer contributions per individual. The complexity of the repository tends to become more stable, with the initial anarchy of complexity decreasing forming a more cohesive and sturdy repo.

The mature stage of a repo shows a further increase in collaborators, being high in numbers but low in commits per individual. Committers joining at this point generally contribute a handful of commits at most, generally observed as pertaining to refactoring and bug fixes; thereby giving extreme stability in complexity changes due to minimal changes to major features.

SLOC v Complexity

SLOC in a software repository refers to the amount of lines of code associated with the source code. While the complexity of a repository tends to reach a steady state after some time due to the repo becoming more mature; sudden large inclusions of source code that then are suddenly removed cause spiking in the relative complexity of a repository. The net effect of this is the addition of churn to the repository. However, as the repository progresses, the given quantities of churn over time gradually decreases in magnitude due to reaching a release. The addition of lines of code continues over time due to collaborators continuing in providing contributions to the repository through bug fixes, and pull requests. But overall, there tends to be a slower growth in the SLOC of a repository relative to the ageing of the project. This is shown in comparing the relative changes in complexity to the additions of code over time to a repository, and the general smoothening out of the contributions' SLOC over time.

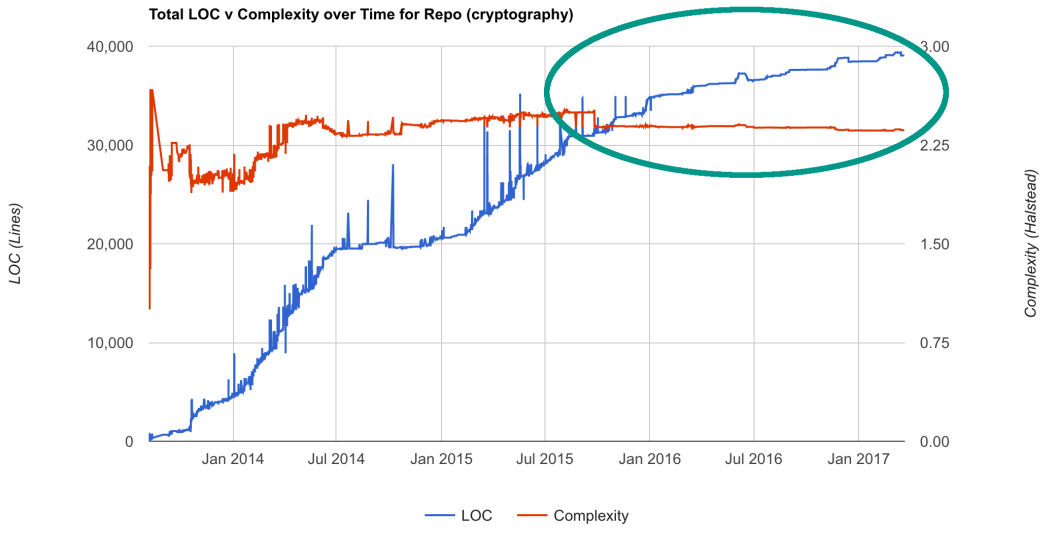


Figure 16 Demarcated mature stage progression

EVALUATION

In the previous chapter, some inferences observed were discussed in the implementation of the toolchain across software repository mining. This chapter sets out to expand on these inferences by providing further discussion and evidence through data set visualisation across mined repositories. The net result is a path of exploration resulting from some interesting relations obtained from a selection of data sets to be further generalised across large data sets in future work.

JOIN-TIME V CONTRIBUTION QUANTITY

Across a wide range of repositories analysed, there appears to be an inversely-exponential snowball effect of the overall individual contribution commit quantity versus the overall swarm size of contributors in an open-source project. The greatest and most frequent contributors throughout the entire life of a project generally tend to be within the first initial groups of joining the repository as a contributor. The effect of this is that the initial contributors tend to become the most predominant contributors overall.

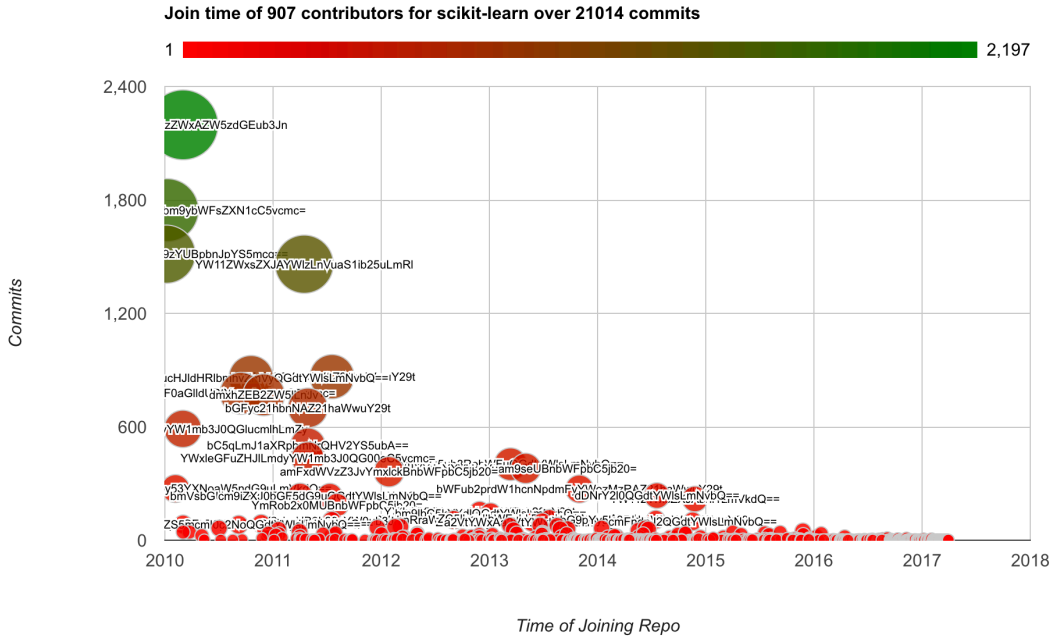


FIGURE 17 JOIN TIME OF COLLABORATORS VERSUS THE CONTRIBUTION SIZE IN COMMITS

However, these contributors have had the highest amount of time in being active members of the project, and so have the effect of time on their side. Later joiners tend to contribute less in comparison to the original contributor group at the start of the project, but there is a much larger quantity of individuals in later groups contributing less individually. The net effect is a small group of contributors being active throughout the life of the project, with vast quantities of collaborators committing pull requests in smaller amounts as per the project's scope.

It appears then, that earlier contributors of a repository tend to become more powerful developers through the earlier adoption of a repository. Throughout the project's life, the initial small core group of developers are constant committers and do the most work individually as per the overall quantity of commits they provide, as pertaining to the scope of socio-metric performance prediction.

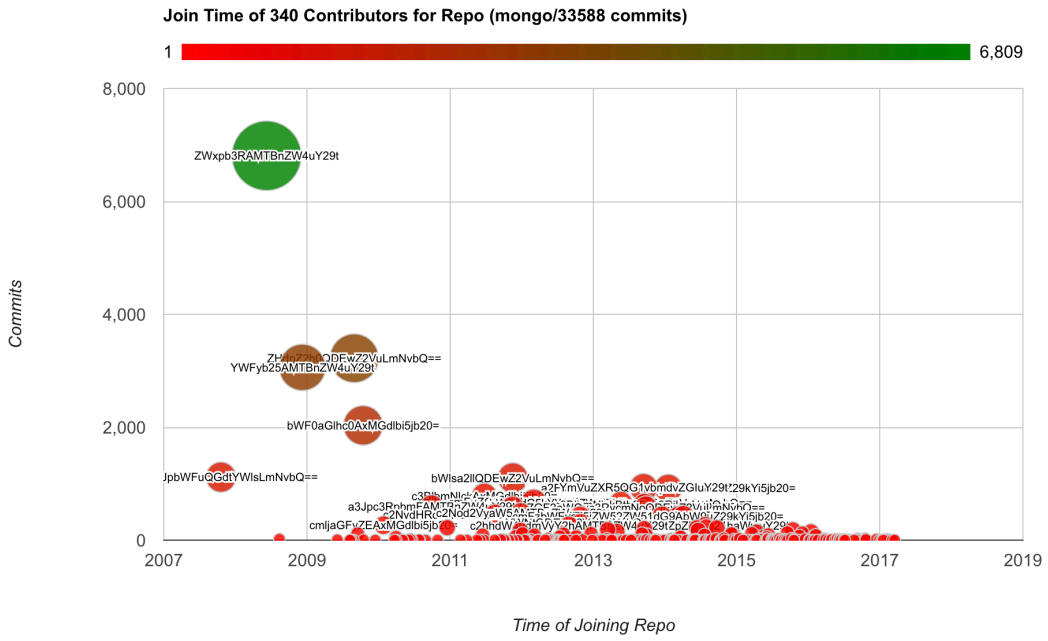


FIGURE 18 MORE CLUSTERING PATTERNS FOR CONTRIBUTORS VERSUS CONTRIBUTION SIZES

An interesting effect here is the visible effect of the Django web framework being developed initially by a core set of developers, and then the handover of the project to the Django Software Foundation – resulting in the dual inverse-exponentiation of the commits versus join time curve.

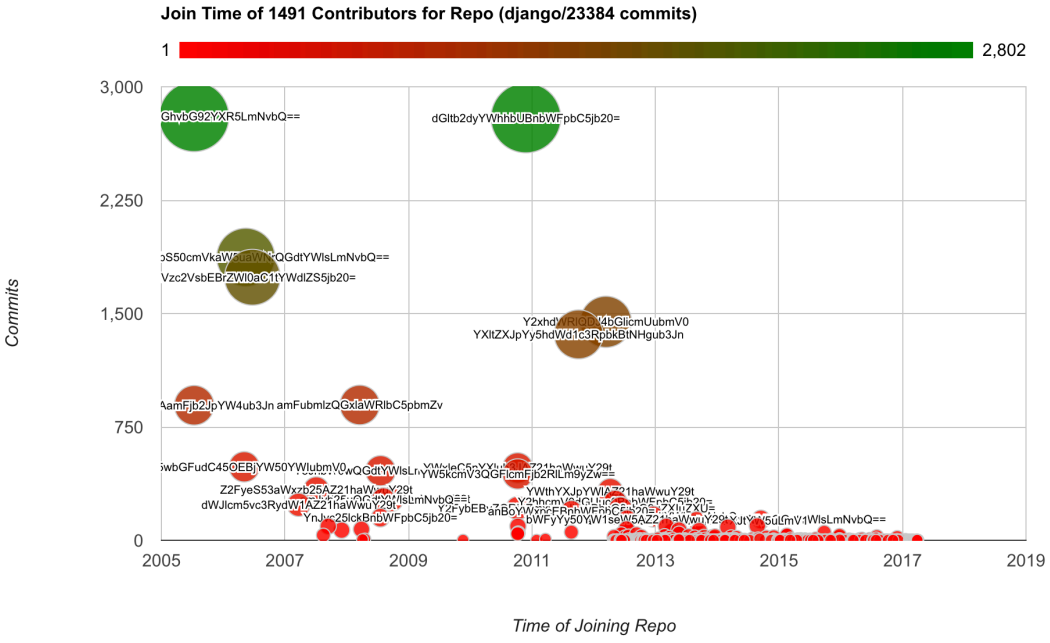


FIGURE 19 AN INTERESTING EFFECT OF A PROJECT HANDOVER BETWEEN CORE DEVELOPERS

Adoption Snowballing v Complexity

The snowballing effect of open-source repositories follows a distinct activation function, where more contributors join a project over time having met a critical mass of interest within the community to spark interest in contributing to the project's goal. As overall goal of open-source software is for a piece of software to make a real difference in the world, it is a place for many novice programmers to gain real-world programming experience by contributing in various ways to support a cause. As later joiners are larger in number, but also contribute fewer commits overall; many of these changes may be bug-fixes or individual changes to the repository to facilitate a change or individual improvement to make the overall system work better overall, such as refactoring work.

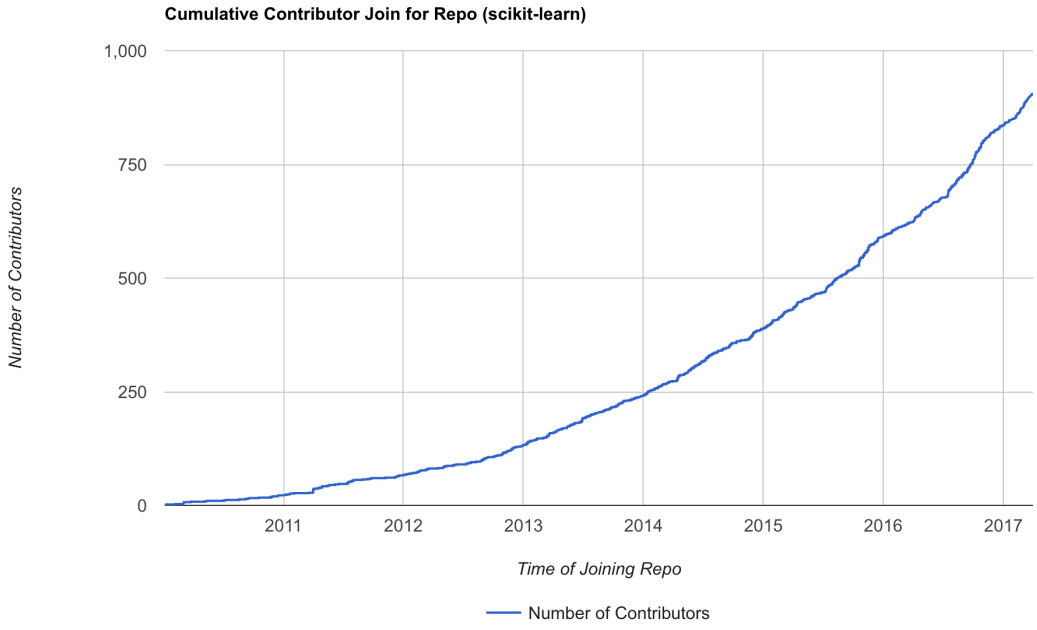


Figure 20 Exponential contributor join

Socio-metrically, the higher the number of collaborators within a repository, the more code styles and individual traits exist within the mesh of code developed by a large group of individuals. As the number of individuals contributing to the repository increases, the code base matures over time, and evolves through the stages of repo maturity. Repositories with a low quantity of contributors do not undergo the same evolution steps as larger repositories do. This infers that as more contributors join a repository, a generic code style as per the repository is developed across developers contributing towards a shared goal.

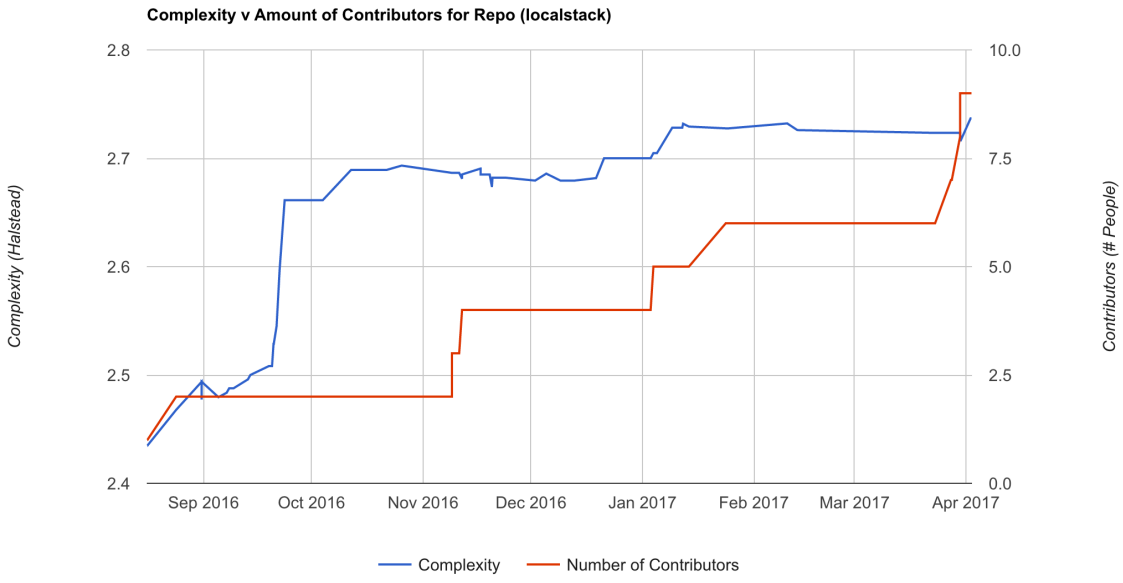


FIGURE 21 A STAGNATING REPOSITORY WITH 8 CONTRIBUTORS

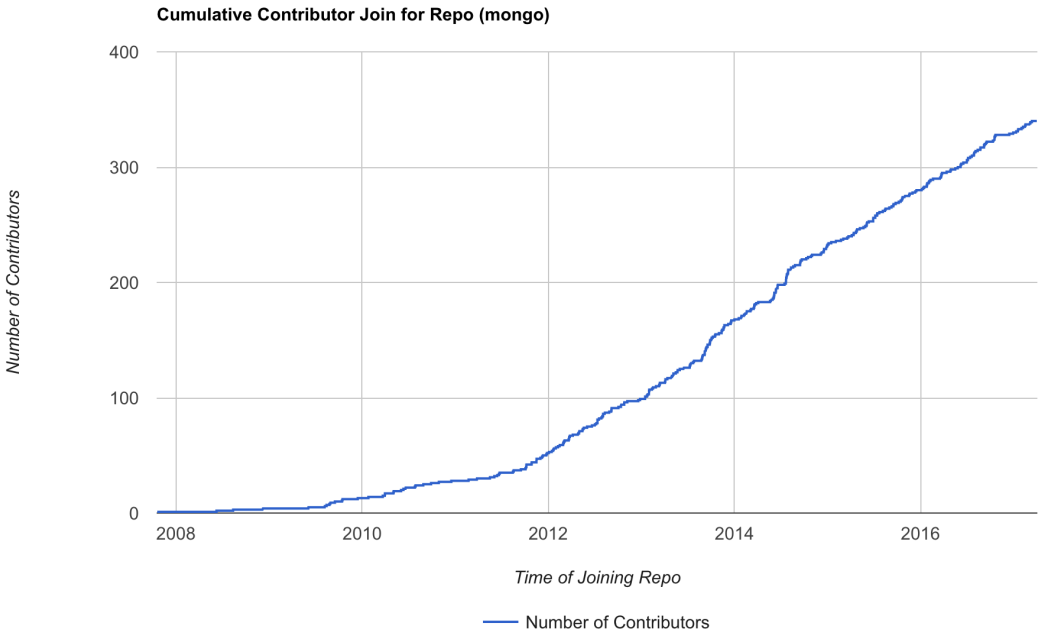


FIGURE 22 DIFFERING JOIN DISTRIBUTION (MONGO)

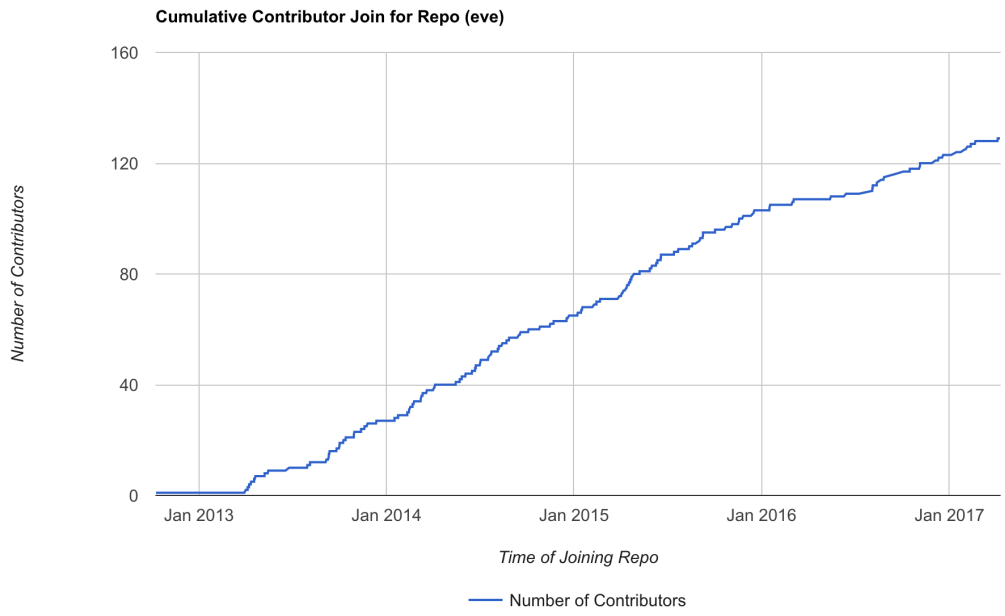


FIGURE 23 ANOTHER COLLABORATOR JOIN DISTRIBUTION (EVE)

STAGES OF REPO COMPLEXITY

Analysis over a wide array of mined repositories tends to show three distinct stages of projects undergoing active development: young, growing, and mature. There appears to be a sudden drop in complexity and then a relatively high stability when the repository reaches a release, once features have been implemented at the software has become generally usable. As the graph below shows, there appears to be a "snowball" effect of the join time of collaborators to a code base, forming an almost negatively exponential curve in the quantity of commits per user joining as the repo progresses in maturity and size.

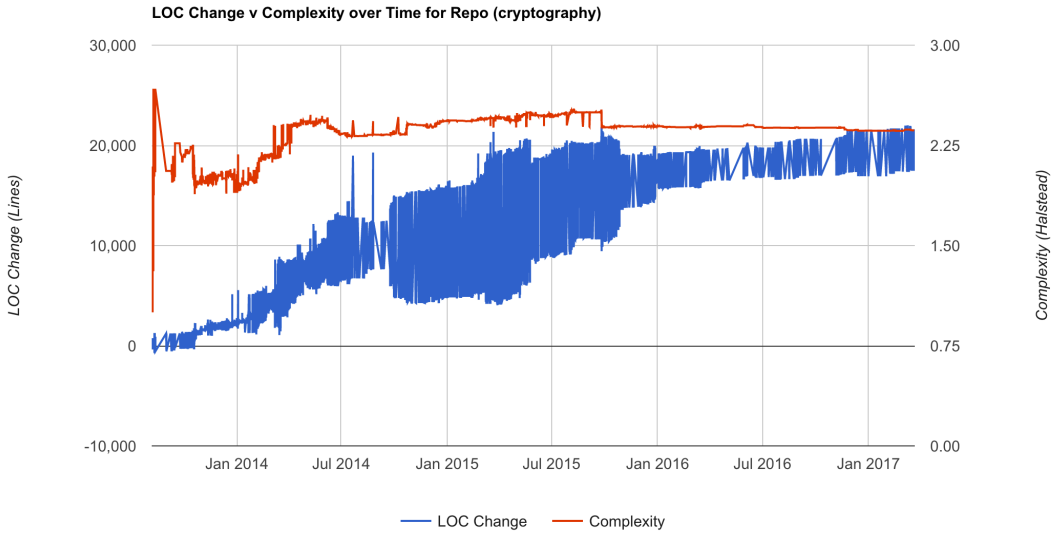


FIGURE 24 DISTINCT AREAS OF DIFFERING LOC CHANGE PER MATURITY LEVEL

There are many interesting properties to explore in the development of a repo. Complexity stagnates over time as the repo ages given a large base of collaborators, resulting in a sudden drop in complexity at a given release. The LOC of code added from this point onwards to the repository no longer spikes upwards sharply at random intervals, and instead tends to stay at the same level over time. Once a release is reached, much of the work associated with new joiners is associated with refactoring work and having open bugs delegated to contributors; rather than that of adding new features to the project.

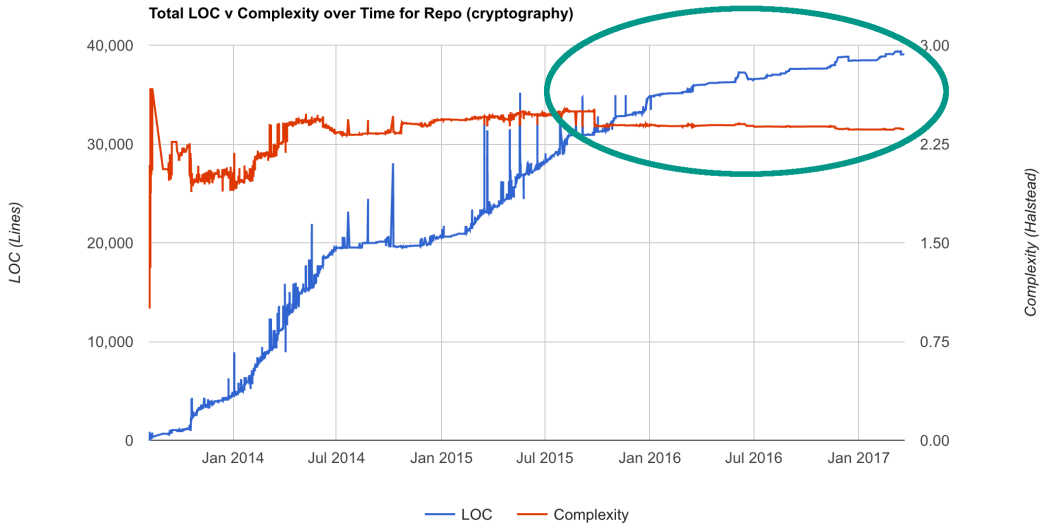


FIGURE 25 DEMARCATED MATURE REPO LEVEL

Across a set of 9 large repositories analysed (cryptography, tornado, keras, tweepy, kivent, flask, treq, eve, Pyrebase) on the next page, all repositories mined underwent three general stages of maturity with a notable drop in complexity between 45-65% of the total percentage of normalised time. These repositories all had differing run times, but do all show similar progressions in complexities over time. Socio-metrically, the stages associated with repositories can be associated with the differing forms of contributors existing at each individual repository stage – earlier contributors tending to work on major features, later contributors focusing on minor features, bug fixes, and refactoring; while documentation also becomes more prevalent throughout the project's life.

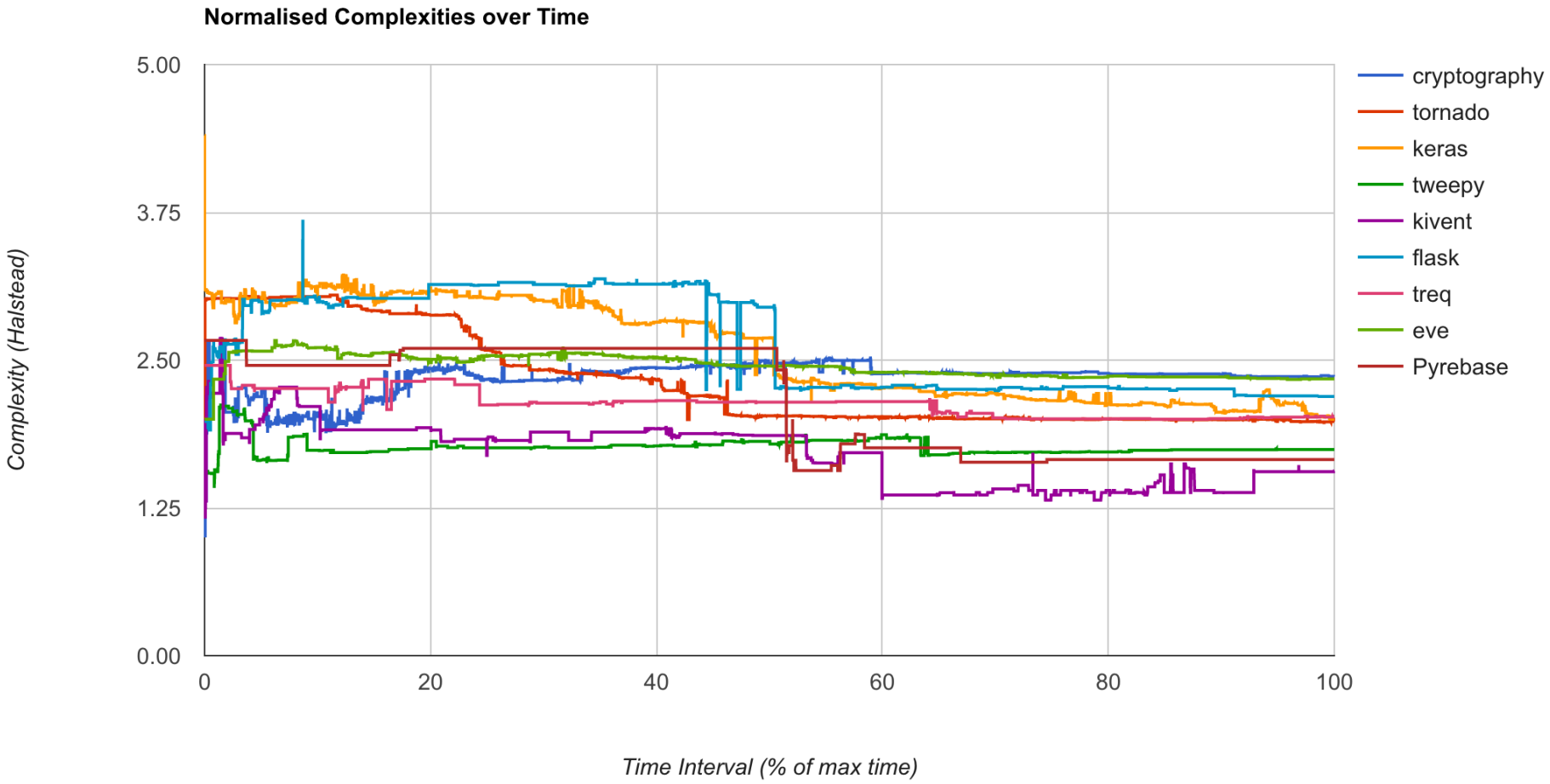


FIGURE 26 NORMALISED COMPLEXITIES OVER TIME ACROSS 9 REPOSITORIES

SLOC v Complexity

A common trait across the most frequent overall collaborators in a repository is that they tend to add the most overall lines of code with respect to the lowest overall summation of changes between commits. The general trend is that the most exposure an individual has towards a code base, the complexity change they demonstrate overall is lower than that of newer collaborators with fewer commits given less overall exposure over time. While this pattern can be seen throughout repositories for the major contributors, there doesn't appear to be a pattern for other smaller contributors.

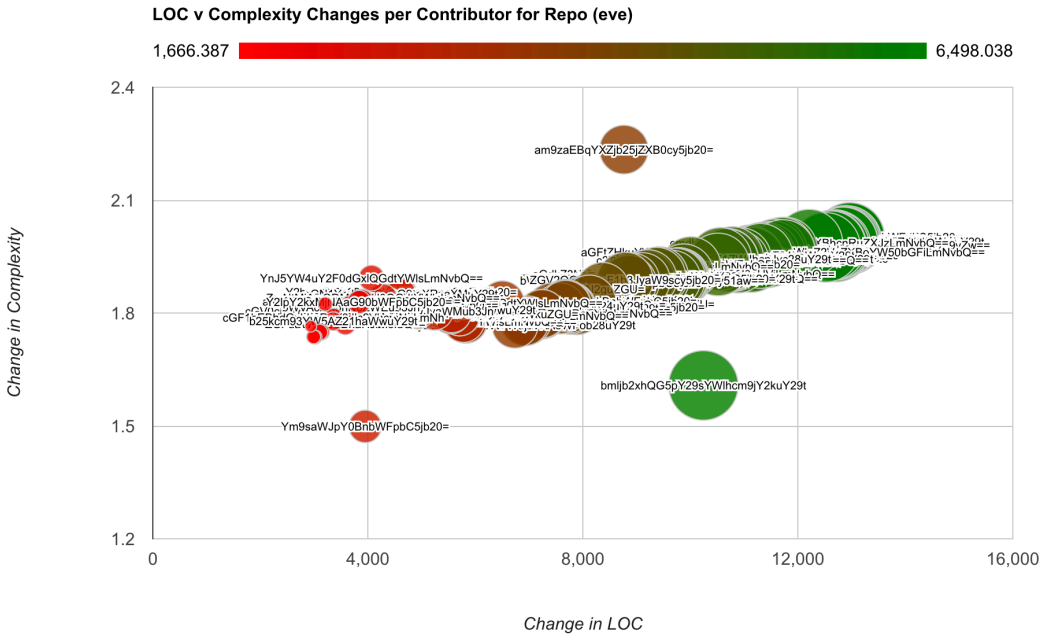


FIGURE 27 HIGHEST CONTRIBUTOR FROM JOIN TIME APPEARING IN THE BOTTOM RIGHT

Complexity, SLOC and churn are related to one another through relative complexity changes within a repository. High amounts of churn give a high variability in complexity and stability. High amounts of churn can indicate uncertainty about requirements, or a highly volatile code base. There is less activity over time as features get implemented and the project reaches the end of the development cycle; and major development work gets more replaced by maintenance, refactoring, and bug fixes.

It appears that this overall summation of differences between commits that a collaborator contributes to the code base and the aggregation of lines of code change to the overall repository, corresponds with the general properties of the maintainability that a contributor provides for its stability. Socio-metrically, this infers that an individual is providing large amounts of contributions to software lines of code, while minimising the impact of adding additional complexities to the repository. On a team scope of socio-metric performance, the greatest contributors as per the join time with respect to commits are always situated towards the bottom right of the graph. Unfortunately, there appears to not be other standard distribution for other less active contributors visible directly inferred across repositories.

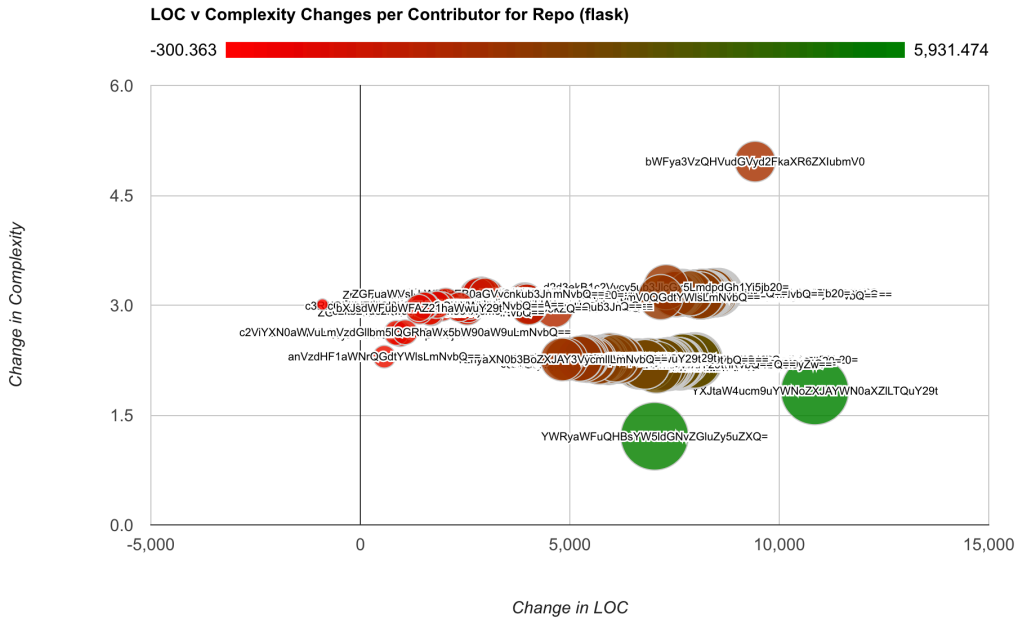


FIGURE 28 TWO HIGHEST CONTRIBUTORS FROM JOIN TIME IN THE BOTTOM RIGHT

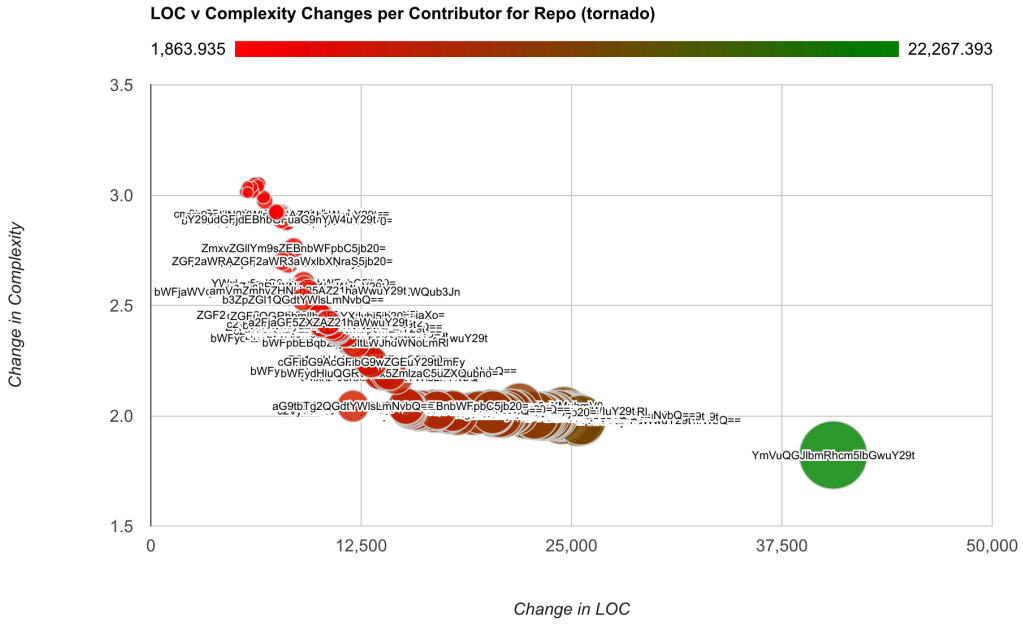


FIGURE 29 HIGHEST CONTRIBUTOR FROM JOIN TIME IN THE BOTTOM RIGHT

Conclusion and Future Work

Intended End-User

On a wider scope, the intended end-user for this platform would be a data scientist interested in generating metrics about the effects and state of a team of developers within an organisation. The platform itself could be integrated as part of a continuous system on a server in tandem with observing the build quality like implementations such as Jenkins, or on a wider scope, and be run over large data sets to compare generalisations of software development to the given development team aggregated over huge amounts of GitHub repositories. Metrics observed from the system infer that certain behaviours result from given engagements and generalise over large data sets over time.

Risk Factors

Verification techniques for this remain as a risk factor, due to the lack of statistical analysis and formality used within the scope of automation. Efforts in this project are based on the nature of software engineering – basing assertions on visualisations obtained through exploration of metrics, and looking generically into applications through it. It should be noted that there are varying degrees of software repositories that are used in different ways: some gradually for building projects, some as a collection of exercises, some pushing an already finished project as a reference, and many other mixtures.

In addition to this, work done within the field is based on only a representative sample of data volumes obtained from digesting software repositories. For these findings to become more concrete, an analysis using a web crawler would have to be run over a very large amount of repository data sets. Given this, running a suite of statistical analysis tools over a large collection of repositories via a web crawler would result in data models being generated on a much more accurate and generalised scope. As the sample size for this project was below 25, the results ascertained from here must be run over a much larger sample size to see if the properties generalise across varying projects.

Given that there are over 26 million users of GitHub alone, and excess of 57 million repositories (GitHub, 2017); the next logical step for statistical certainty and verification within the realm of software engineering is to expand the project by building a web crawler for running analysis across huge data-sets of repositories. However, minimal statistical certainty for any repository generalisations would require a sample size of 16,500 repositories at 99% significance given a 1% margin of error to justify any potential relationships and generalisations across data-sets at scale. Given scaling, a generalisation of patterns observed can be explored in greater detail as part of qualitative research. It has been demonstrated though that tool-set is possible through sets of engineering challenges portrayed in this project, and expansion to further the overall goal of scale requires further development for it to be possible to obtain larger data sets.

In any case, there is an extremely high dimensionality associated with software repository meta data in accordance with engineering. The repercussion of this is that data-sets may be too complex to accurately define correlations, and may inevitably have too many correlations may exist between data sets for a researcher to accurately deal with.

Future work

There is a share of worthwhile consideration of consider for future work within the scope of this field and project. It was to the best of the author's knowledge given the time of planning and

implementation, with only a given amount of experience within the creation and deployment of such a system as mentioned below.

Verification Techniques

Section 2 outlined that the statistical analysis for this project was based primarily upon drawing upon a wealth of literature given exploratory techniques between various derived metrics as per Halstead, McCabe and Maintainability Index. There is a potential to expand on these known paradigms of metric generation and provide more meaningful insights and generate further validity.

Efforts have been focused primarily on human-code socio-metrics not strictly tied to a certain paradigm. In the future, further work to be completed may pertain to examining the effects of various code paradigms on the overall relative complexity within a software repository in addition to the effects of modularity and language-specific contexts. The tool-chain should be expanded upon, and build upon literature of modular and non-modular designs with the effects of fan-in and fan-out on the overall complexity – such as object-oriented, functional, symbolic, and procedural to name a few.

Conclusion

The overall aim of the project was to design and implement a suite of tools to allow and satisfy the design goals of exploring human-code metrics as defined in chapter 2. The system as per the tool-chain satisfied all pre-conditions objectives for experimentation of socio-metric code analysis as defined in chapter 1. The toolchain has also succeeded in pertaining to the preconditions as per observed repository complexity, contribution effects and differences between collaborator efforts, join times, and the outcome of ranking the individual efforts of an individual towards the overall scope of the open-source project's goal as per chapters 3 and 4, with further discussion in chapter 5 over a greater generalisation of project sizes.

It is interesting to compare the effects of varying methodologies of quantifying and qualifying the individual effects of a collaborator's contributions towards a repository as proposed by Halstead and McCabe and contrast this with the real-life observed complexities on a wide range of open-source projects over a large time interval with real-life developers.

This project has explained the shortcomings of a relatively small data set, but has also uncovered interesting metrics associated with the long-term development practices. The differences observed between project contributor sizes is also an interesting metric to observe the changes in development styles according to the team size. It has been interesting to note the socio-metric performance differences between the aggregation of project stages and complexities over time that follow with larger numbers of contributors in a project in comparison. Smaller sized projects tend not to undergo the same levels of maturity and agnosticism of style meshing over time, and so tend not to reach a complexity stagnation over time.

It can therefore be concluded that the advantages of analysing repositories for team and individual socio-metric performance metrics can be further investigated to reveal more in-depth data about the trends observed within a repository, and reveal the true traits of main contributors within open-source software repositories.

BIBLIOGRAPHY

Atlassian, n.d. *What is Git.* [Online]
Available at: https://www.atlassian.com/git/tutorials/what-is-git
[Accessed 27 April 2017].

Booyabazooka, 2006. *Directed graph. graph { 1 -> 2; 2 -> 1; 2 -> 3; 3 -> 2; }.* [Online]
Available at: https://upload.wikimedia.org/wikipedia/commons/a/a2/Directed.svg
[Accessed 27 April 2017].

Borysowich, C., 2007. *Design Principles: Fan-In vs Fan-Out.* [Online]
Available at: http://it.toolbox.com/blogs/enterprise-solutions/design-principles-fanin-vs-fanout-16088
[Accessed 27 April 2017].

Celis, I., 2010. *Custom event emitters in Javascript.* [Online]
Available at: https://robots.thoughtbot.com/custom-event-emitters-in-javascript
[Accessed 27 April 2017].

cirosantilli, 2016. *all-github-commit-emails.* [Online]
Available at: https://github.com/cirosantilli/all-github-commit-emails
[Accessed 27 April 2017].

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C., 2001. *Introduction to Algorithms.* s.l.:MIT Press and McGraw-Hill.

Dissanayake, P., 2014. *How to draw a Control flow graph & Cyclometric complexity for a given procedure.* [Online]
Available at: https://geekdetected.files.wordpress.com/2013/03/untitled.jpg
[Accessed 27 April 2017].

ECMA, 2013. *The JSON Data Interchange Format.* [Online]
Available at: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
[Accessed 27 April 2017].

Even, S., 2011. *Graph Algorithms.* s.l.:Cambridge University Press.

Fontana, F. A., Braione, P. & Zanoni, M., 2011. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology,* 11(2), pp. 1-38.

Freeman, P. & Hart, D., 2004. A Science of Design for Software-Intensive Systems. *Communications of the ACM,* 8 August, 47(8), pp. 19-21.

GitHub, 2017. *Celebrating nine years of GitHub with an anniversary sale.* [Online]
Available at: https://github.com/blog/2345-celebrating-nine-years-of-github-with-an-anniversary-sale
[Accessed 27 April 2017].

GitHub, 2017. *How we share the information we collect.* [Online]
Available at: https://help.github.com/articles/github-privacy-statement/#how-we-share-the-information-we-collect
[Accessed 27 April 2017].

Goodrich, M. T. & Tamassia, R., 2001. *Algorithm Design: Foundations, Analysis, and Internet Examples.* s.l.:Wiley.

Google, 2016. *MVC Architecture.* [Online]
Available at: https://developer.chrome.com/static/images/mvc.png
[Accessed 27 April 2017].

Hakes, T., 2016. *GitHub Exposes Your Personal Email Address.* [Online]
Available at: https://taylorhakes.com/posts/get-any-github-users-email-address/
[Accessed 27 April 2017].

Halstead, M. H., 1977. *Elements of Software Science.* Amsterdam: Elsevier Science Ltd.

Harrison, W. A., 1984. *Applying McCabe's complexity measure to multiple-exit programs.* s.l.:John WIley & Sons Ltd.

High Scalability, 2016. *A Beginner's Guide To Scaling To 11 Million+ Users On Amazon's AWS.* [Online]
Available at: http://highscalability.com/blog/2016/1/11/a-beginners-guide-to-scaling-to-11-million-users-on-amazons.html
[Accessed 27 April 2017].

IEEE, 1990. *610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology.* New York(New York): IEEE Computer Society Press.

McCabe, T. J., 1983. *Structured Testing.* s.l.:IEEE Computer Society Press.

Naboulsi, Z., 2011. *Code Metrics – Maintainability Index.* [Online]
Available at: https://blogs.msdn.microsoft.com/zainnab/2011/05/26/code-metrics-maintainability-index/
[Accessed 27 April 2017].

Nguyen, V., Deeds-Rubin, S., Tan, T. & Boehm, B., 2007. *A SLOC Counting Standard,* s.l.: University of Southern California .

Paul, R., 2006. *Surveys show open source popularity on the rise in industry.* [Online]
Available at: https://arstechnica.com/uncategorized/2006/01/6017-2/
[Accessed 27 April 2017].

Podjarny, G., 2015. *Keeping your Open Source credentials closed.* [Online]
Available at: https://snyk.io/blog/leaked-credentials-in-packages/
[Accessed 27 April 2017].

Radon, n.d. *Introduction to Code Metrics.* [Online]
Available at: http://radon.readthedocs.io/en/latest/intro.html
[Accessed 27 April 2017].

Reenskaug, T. & Coplien, J. O., 2009. *The DCI Architecture: A New Vision of Object-Oriented Programming.* [Online]
Available at: http://www.artima.com/articles/dci_vision.html
[Accessed 27 April 2017].

Serebrenik, A., 2011. *Software Evolution.* [Online]
Available at: http://www.win.tue.nl/~aserebre/2IS55/2010-2011/10.pdf
[Accessed 27 April 2017].

Sink, E., n.d. *Git: Cryptographic Hashes, DVCS Internals.* [Online]
Available at: http://ericsink.com/vcbe/html/cryptographic_hashes.html
[Accessed 27 April 2017].

Thompson, B., 2016. *6 Causes of Code Churn and What to Do About Them.* [Online]
Available at: https://blog.gitprime.com/6-causes-of-code-churn-and-what-to-do
[Accessed 27 April 2017].

VerifySoft, n.d. *Halstead Metrics.* [Online]
Available at: http://www.verifysoft.com/en_halstead_metrics.html
[Accessed 27 April 2017].

VirtualMachinery, n.d. *The Halstead Metrics.* [Online]
Available at: http://www.virtualmachinery.com/sidebar2.htm
[Accessed 27 Apri 2017].

Watson, A. H. & McCabe, T. J., 1996. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.* Gaithersburg(Maryland): NIST.