Investigating the Utility of Agile and Lean Software Process Metrics for Open Source Software Communities: An Exploratory Study

by

Disha Suresh

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2022 by the
Graduate Supervisory Committee:

Kevin Gary, Chair
Srividya Bansal
Alexandra Mehlhase

ARIZONA STATE UNIVERSITY

December 2022

ABSTRACT

The adoption of Open Source Software (OSS) by organizations has become a strategic need in a wide variety of software applications and platforms. Open Source has changed the way organizations develop, acquire, use, and commercialize software. Further, OSS projects often incorporate similar principles and practices as Agile and Lean software development projects. Contrary to traditional organizations, the environment in which these projects function has an impact on process-related elements like the flow of work and value definition. Process metrics are typically employed during Agile Software Engineering projects as a means of providing meaningful feedback. Investigating these metrics to see if OSS projects and communities can utilize them in a beneficial way thus becomes an interesting research topic. In that context, this exploratory research investigates whether well-established Agile and Lean software engineering metrics provide useful feedback about OSS projects. This knowledge will assist in educating the Open Source community about the applications of Agile Software Engineering and its variations in Open Source projects. Each of the Open Source projects included in this analysis has a substantial development team that maintains a mature, well-established codebase with process flow information. These OSS projects listed on GitHub are investigated by applying process flow metrics. The methodology used to collect these metrics and relevant findings are discussed in this thesis. This study also compares the results to distinctive Open Source project characteristics as part of the analysis. In this exploratory research best-fit versions of published Agile and Lean software process metrics are applied to OSS, and following these explorations, specific questions are further addressed using the data collected. This research's original contribution is to determine whether Agile and Lean process metrics are helpful in OSS, as well as the opportunities and obstacles that may arise when applying Agile and Lean principles to OSS.

ACKNOWLEDGMENTS

First, I would like to sincerely thank my thesis advisor Dr. Kevin Gary for keeping me motivated throughout my research tenure through constant guidance and support. He not only guided me through regular technical discussions and meetings, but also cultivated a sense of ownership and innovation by sharing several valuable ideas. The various aspects of statistical analysis and data querying would not have been possible without Dr. Gary's expertise and the numerous hours he put in to augment my work.

I would like to express my sincere appreciation to my thesis committee members – Dr. Srividya Bansal and Dr. Alexandra Mehlhase, for taking their valuable time out and kindly agreeing to attend my thesis defense. Thank you for taking interest in my research.

With much love and gratitude, I want to thank my parents and friends for providing constant support and encouragement throughout this journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Software development is the process of building a software program designed to perform a specific task [1]. Given the tremendous shift in the adoption of software by traditional businesses, the software development industry has perhaps the fastest growth rate [2]. A process or set of processes used in software development is referred to as software development methodology. Over the past 15 years, Agile has raised itself to the top of software development methodologies [3]. The Agile Manifesto [4] describes Agile as adaptive and iterative, in contrast to Waterfall's sequential and non-iterative approach [5, 6]. The very foundation of Agile development is the expectation that both the processes and the final product will evolve in response to customer feedback. Agile methodology has a fair amount of overlap with Lean [7, 8] as both the product and process are continuously improved through learning cycles and development. According to 15th Annual State of Agile Report [9], Agile adoption within software development teams increased from 37% in 2020 to 86% in 2021. With increased popularity, more organizations are being drawn to Agile. However, in order to effectively apply Agile, it is important to understand how to evaluate its success. This can be accomplished by using Agile metrics as they provide insight that helps in evaluating the quality of a product and monitoring team performance [10]. Although quality metrics are crucial, in this study emphasis is laid on the Agile/Lean process metrics that use the flow of work data to offer insightful information.

Open Source Software (OSS) [11, 12] has become the foundation for almost all modern software today. According to Forrester Research in 2017 [13], only 10 to 20% of new application code within a corporation is proprietary and the remainder is open source.

OSS has gained enormous popularity and organizations are becoming more aware of the benefits and flexibility that can be obtained by utilizing and funding it. According to the survey from The New Stack in 2021 [14], 63% of organizations that use OSS said that these programs are incredibly valuable to the success of their engineering and product teams. OSS communities also have tremendous economic impact. $55 billion was spent on Open Source mergers and acquisitions in 2018 alone [15].

OSS initiatives frequently share the same values and principles as Agile and Lean software developmental projects [16]. Some of these shared principles include the emphasis on people over process, frequent communication, building product around highly skilled personnel at the heart of a self-organizing development team, the acceptance of change by employing short feedback loops with frequent releases of functional code, and the close collaboration with clients and users [4, 17]. On the other hand, there are some key distinctions between them and commercial organizations, such as the lack of rigid release deadlines, typically relying on resources from volunteers and part-time personnel, and the ability to transition over time in ways that are different from commercial organizations. The team co-location demanded by Agile development is not seen as a precondition in Open Source development as they are usually distributed [17]. Therefore, examining the effectiveness of Open Source communities is worthwhile. The first step in achieving that is to determine whether or not the ways we use to measure success using Agile/Lean process metrics can be applied to OSS.

In this exploratory study, well established Agile/Lean software process metrics are applied to OSS development projects. Following these investigations and observations, specific questions are further answered utilizing the data collected. The primary contribution of this research is to investigate the usefulness of Agile/Lean process metrics in OSS, as well as the potential benefits and drawbacks of adopting Agile and Lean principles to OSS.

The rest of the thesis is divided into 6 chapters. Chapter 2 focuses on literature review related to this research. The first section emphasizes the study of various well-established software process metrics, and their importance in the field of software engineering. The second portion of the chapter focuses on the literature and characteristics of OSS initiatives. Research questions, research methodology, and techniques of data collection used in this study are described in Chapter 3. Based on their relevance to the OSS ecosystem, OSS projects and characteristics, as well as Agile/Lean software development metrics included in this study is discussed in this chapter. The design and implementation of this thesis are discussed in Chapter 4. It focuses on gathering all the essential data from GitHub and integrating it with a report generator to assist visualize the utility of Agile/Lean software process metrics. As each of the shortlisted metrics is analyzed against the previously chosen OSS projects, Chapter 5 summarizes the observations of what these metrics tell us. This chapter also focuses on the exploratory queries that emerge from the findings of our study's visualizations. For a few of these queries, further drill downs are made to investigate if the Agile/Lean development process metrics provide any insight into how OSS communities operate. Chapter 6 focuses on the the implications and limitations of using the Agile/Lean metrics in the context of OSS development communities. Finally, Chapter 7 summarizes the technical findings of this work, along with the future scope in this research.

CHAPTER 2

LITERATURE REVIEW

This chapter specifically focuses on literature related to this research. The first section emphasizes the literature that establishes various important metrics, and their importance in the field of software engineering. The second section focuses on the literature available for OSS projects and its characteristics.

2.1 Agile and Lean Metrics

Software metrics can be used to acquire a great deal of information such as the scale and complexity of a software system as well as the quality of the product that is provided to the client along with the progression of a software project [18]. Product, process, and project metrics are the three categories of software metrics involved in the traditional software development process [19]. Product metrics are used to define a product's attributes, such as its size, complexity, design features, performance, and level of quality [20]. To optimize software development and its maintenance, process metrics can be used. Examples include the efficiency with which defects are eliminated during development, the pattern of testing defect occurrence, and the turnaround time of the fix process [21]. Project execution and characteristics are described by project metrics [19, 22]. Examples include the quantity of software developers, the pattern of staffing across the software's life cycle, cost, schedule, and productivity. This research mainly focuses on process metrics and the flow of work in a project.

One of the fundamental issues that arise during the software development process is change in requirements. As a result, while choosing these software process metrics, care must be taken to fully comprehend the iterative and incremental development process, which is flexible to accepting changes [18]. Agile Software Development (ASD) effectively manages the reality of change unlike traditional software development. Further, the limitations imposed by traditional metrics [19] such as planned capacity and schedule variance regarding their flexibility in changing requirements hamper their suitability for ASD. Thus, it is crucial to find a set of process metrics that are better suited for the ASD given that adaptation into the ASD process is fast growing.

The key to a successful metric strategy is to identify the right set of metrics that provide the information to act and support decision making [23]. For the past 30 years, traditional software development metrics have been used in various practices. Booch [24] introduced software metrics for process improvement and project management in 1992. Putman et al. [25] identifies size, productivity, time, effort, and dependability as the five core parameters necessary for successful software development. The study in Pulford et al. [26] provides the following motivations for the usage of metrics: a) planning and estimation of a project, b) managing and monitoring a project, c) identifying quality corporate goals, and d) improved processes, tools, and communication in software development. Although this set of core measurements are still relevant, the increasing scope of research and interest in the field of Software Engineering facilitated the inception of several important metrics to this list.

Oza and Korkala [27]] characterizes the indicators utilized in the ASD process as code level, productivity/effort level, and economic metrics. The code-level metrics attempt to provide insight into the quality of the code. For instance, running tested features, Leffingwell's [79] iteration and release perspectives, and code quality and design metrics. The economic metrics such as earned business value and break-even point help with decision-making. To track team progress, productivity measures like burn-down charts and project size units are useful. Eventually, these metrics are classified into seven categories including strategic, testing, iteration, automation, code, engineer, and project management. While the categorization of these metrics was done in consideration of ASD practices, the proposed research highlights how their applicability is equally relevant to OSS projects.

Although Agile and Lean development organizations appear to have compelling reasons for needing metrics, it is unclear which metrics Agile teams are utilizing in reality, why they are using them, and how they are benefiting from it. Agile and Lean are examined together since a comparison shows that they have similar objectives and rely on similar underlying ideas [28].

The concept of waste reduction is fundamental to the optimization of all operations that produce inefficiencies. Lean software development [29] solves this problem by improving the development processes. This point of view complements Agile principles by focusing on all activities that add value to the customer. Lean concepts described in Poppendieck et al. [30] can be applied to scale-up Agile software development processes or to adapt already-existing Agile practices. In that context, Kišš et al. [31] review and categorize the following seven metrics that have been used to measure the Lean transformation: Lead time, Number of defects, Fix time for defect, Velocity, Lines of Code, Story rate per iteration and Release Frequency. These metrics can be broadly categorized as either relating to processes or to quality. The proposed study concentrates on measures that are relevant to processes, such as Lead time, Velocity, Story rate per iteration, and Release Frequency.

In his book *Project to Product* [28], Mik Kersten proposes the Flow Framework. This framework delineates four flow items: features, defects, risks, and debts. The proportional time spent on each type of flow item is the flow distribution. Flow metrics are used to measure the flow in the system, with each representing an intersecting perspective on the nature of value stream flow in software development. Of these metrics in the Flow Framework, Kersten states that Flow Time is the most meaningful, and that alternate interpretations of Lead Time (namely the lifecycle of an issue) are not as important. On the contrary, in my work I will demonstrate that both these measures are meaningful in the context of OSS development teams.

K. V. J. Padmini et al. [18] conducted survey and interview-based analysis of 24 development companies in their work, in which they identified metrics that could be beneficial to the ASD process. In their study, they describe how software metrics proved to be useful for forecasting projects and project management, keeping track of quality of product, and reviewing progress of a project. Among those the top ten metrics are Delivery on time, Work capacity, Unit test coverage for the developed code, Percentage of adopted work, Bug correction time from new-to-closed state, Sprint-level effort burndown, Velocity, Percentage of found work, Open defect severity index, Focus factor, and Cost of quality. However, after examining the correlation between the presented research and this study, it became clear that several of these metrics needed to be redefined in the context of OSS, the study's main focus.

According to Kupiainen et al., [32], software metrics are employed to achieve clear objectives such as project planning and regulating the running of sprints, reviewing project progress, understanding, and increasing quality, identifying solutions to challenges, and encouraging teams. In general, software metrics are used to characterize project performance by tracking team members' engagement intensity, improving communication among team members, and measuring software quality. They use both qualitative and quantitative perspectives to highlight the importance of these high influence metrics.

Downey and Sutherland [33] identify nine essential metrics which are meaningful and can be used for managerial decision making. Those metrics include Velocity, Work Capacity, Focus Factor, Percentage of Adopted Work, Percentage of Found Work, Accuracy of Forecast, Targeted Value Increase (TVI+), Success at Scale, and Win/Loss Record. The authors contend that these metrics are more valuable than Story Points because they allow management to compare the performance of many teams, rather than just the performance of the original team. Given that all of the proposed metrics in this work are utilized to improve project's flow efficiency, the topics discussed in this work are pertinent to the current research's field of study.

Based on 13 selected studies, relevant Scrum software metrics for each of the four scrum events (sprint planning, daily scrum meetings, sprint review and sprint retrospective) are highlighted in Kurnia et al. [34]. The study by Sambinelli et al. [35] proposes a set of customer values metrics including Number of escaped defects, Work in progress, Cycle time, Business value points, Points for user history, Cost, Average time to stabilize a software, Benefit or cost performance index and Function points. The proposed study contrasts with this work by categorizing Escaped Defects and Work in Progress as Lean metrics that are predominately used to measure waste since both metrics provide little explanation about the value perceived by the customers.

Since the main goal of the Lean methodology is to reduce and eliminate waste in non-value-adding operations, it constantly highlights the value of quality. According to Deming [36], we must uphold a high standard of quality all through the process cycle. To continue focusing on initiatives that aim for continuous improvement [37] and to quantify the organization's progress, quality metrics are required. Escaped Defects, Cyclomatic Complexity, Comment Percentage, Size, Cohesion, and Coupling are some of the crucial quality metrics. However, for the sake of this study, we are more interested in metrics that are associated with the process. Table 2.1. provides a summarized list of Agile and Lean metrics in relevant state of the art studies.

Table 2.1: Review of Agile and Lean Metrics in Relevant State-of-the-art Studies

Authors	**Agile and Lean Process Metrics**
Kišš et al. [31]	Lead time, Number of defects, Fix time for defect, Velocity, Lines of Code, Story rate per iteration and Release Frequency.
Kupianinen et al. [32]	Velocity, Work in progress, Lead time, Burndown chart, Cycle time, Effort estimate, Story percent complete, Queue time, Processing time, Check-ins per day, Variance in handovers, Deferred defects, Predicted number of defects in backlog, Test coverage, Test-growth ratio
Kurnia et al. [34]	Velocity, Story point, Sprint burndown, Release burnup, Value delivered, Customer satisfaction, EVM, Job satisfaction
Sambinelli et al. [35]	Number of defects per period, Work in progress, Cycle time, Business value points, Points for user history, Cost, Average time to stabilize a software, Benefit or cost performance index, Function points
Padmini et al. [18]	Delivery on time, Work capacity, Unit test coverage for the developed code, Percentage of adopted work, Bug correction time from new-to-closed state, Sprint-level effort burndown, Velocity, Percentage of found work, Focus factor, and Cost of quality.
Downey and Sutherland [33]	Velocity, Work Capacity, Focus Factor, Percentage of Adopted Work, Percentage of Found Work, Accuracy of Forecast, Targeted Value Increase, Success at Scale, and Win/Loss Record.
Mik Kersten [28]	Flow Velocity, Flow Load, Flow Time, Flow Efficiency, Flow Distribution
Donald G. Reinertsen [74]	Cumulative Flow Diagram

2.2 Open Source Software and Its Characteristics

Open Source Software (OSS) has become crucial to modern software engineering [38,39]. Numerous quantitative analyses have demonstrated that utilizing OSS is frequently a rational strategy than proprietary alternatives [40]. The history of Open Source may be traced back to the creation of computer software. Early software, albeit not known as Open Source at the time, was freely shared among developers and not regarded as a commercial product [41]. In the present, the Open Source Initiative provides a 10-point Open Source Definition [42], a detailed definition giving ten criteria that a license must comply with to be recognized as Open Source. The major components of this definition include a) The instructions for running the software are contained in the lines of code that make up the source code. A person often needs access to the source code if they want to modify a piece of software; b) Without paying a fee or a royalty, a person may use all or parts of OSS as a component in another, larger software program; c) OSS may be modified or expanded, and the newly produced software may be distributed.

The idea of Open Source also implies a team-based method of invention where a software program is created by a virtual team of programmers. The team members often participate in projects as volunteers without being necessarily employed in the same organization [43]. Open Source communities typically form and grow organically [44], adhering to governance systems that are frequently only loosely defined [45, 46].

Not all Open Source initiatives are founded in a community. They could also be started and managed by an organization. It is the firm’s challenge to get outside developers to work on a project that the company is leading [47]. Many organizations also compensate staff members for their contributions to both company-led and community-led projects, as is the case with IBM's participation in the Apache project. In addition to getting free testing and enhancement advice from volunteers, businesses also save money by employing OSS [48]. Organizations are an active player in the Open Source movement [49]. More than a quarter of OSS engineers received compensation from a for-profit organization, according to a thorough survey in 2003 [50].

It is important to understand that large projects frequently have a small core team who dedicate significantly both in time and output [51]. According to surveys, developers devote more than 25% of a typical work week—11 hours on average per week, or a median of 7 hours—to Open Source work [50,52]. Furthermore, programmers who are employed full-time to contribute will have participation rates that are above average. However, the bulk of programmers are infrequent contributors, and their contributions may be made with little to no work.

One must gain a deeper understanding of the Open Source phenomenon before examining how process metrics apply to OSS. There is a need to understand the differentiating characteristics that set OSS projects apart from the traditional software engineering projects. The significance of Open Source characteristics has been covered in several studies in the past. Tamburri et al. [53] studies the community regulatory aspects behind Open Source work and proposes YOSHI (Yielding Open Source Health Information), a tool that can plot Open Source communities against observable community patterns, groups of well-known organizational and social structure types, and traits with quantifiable core properties. The present effort has been motivated by this study's robust model for comprehending the metrics in OSS projects. YOSHI's attempt to describe a community is significant as it influences how one evaluates the flow of work. The 6 characteristics computed in the study are: community engagement, community structure, community formality, community longevity, community cohesion and community geographical dispersion.

According to Gezici et al. [54], there are five distinct types of OSS success criteria: market success, developer activity, quality, organization friendliness, and adaptation. The findings demonstrate that most of the time, OSS success is ultimately regarded as market success. Additionally, the authors assert that developer involvement is just as useful as market success. This claim is supported by most of the success metrics that include the size of contributors.

DeLone and McLean in their comprehensive model [55] for information system (IS) success suggest six measures of success: system quality, information quality, use, user satisfaction, individual impact, and organizational impact. However, in an OSS context, some of the measures are inapplicable, while others are challenging to implement. Many of these metrics do not factor in the unique characteristics of the OSS development. DeLone and McLean's success model is reexamined in Crowston et al. [56] to find additional measures for OSS project success. The authors identified Project output, the Process of Systems Development and Outcome of project members as the new measures of success.

Capiluppi et al. [57] presents a horizontal study aimed at characterizing OSS. The study examined a representative sample of about 400 projects from a well-known OS project repository. Numerous characteristics are defined for each project. However, the attributes related to product quality are not included as it deviates from the scope of this study. Some of the characteristics described in the study are age, application domain, programming language and size of the source code, developer community, number of users who use the code along with population and vitality of the project. While some of these metrics can be consolidated into a single characteristic, some of the others need to be changed to meet the needs of the OSS communities as they currently stand.

Ewusi-Mensah [58] states how the mere completion of a project might be taken as an indicator of success, considering the high number of abandoned projects. But it is hard to define when OSS projects are finished because many of them are constantly under development. In response to this issue, Crowston et al. [56] propose that success can instead be assessed when a project transitions from alpha to beta phase. The interval between releases is another metric relevant to community activities. The phrase "release early and release often" [59] has become a well-established norm in Open Source development, suggesting that an active release cycle is an indication of a sound development method and project.

The literature discussed in this chapter introduced concepts like Agile/Lean metrics and characteristics of OSS that play an important role in this thesis. The papers discussed in the literature provide an explanation of the significance of metrics in measuring Agile/Lean software development success. In contrast to the papers mentioned above, this research focuses on the use of Agile/Lean metrics for OSS. The context for this research is given in the following chapter. It provides a summary of the research questions that guide this thesis work and briefly describes the methodology. Additionally, it provides a brief summary of the problem and the validation process.

CHAPTER 3

RESEARCH CONTEXT

The purpose of this exploratory study is to determine whether adopting the same metrics employed by Agile and Lean communities can enhance and increase the efficiency of OSS processes. Our analysis starts by selecting projects that we considered to be large and likely to have a tracked flow of work. There are millions of Open Source projects on GitHub. Population sampling is not the focus of this methodology. It is crucial to remember that this is not an exhaustive list, merely a representative one. On the other hand, from a metrics viewpoint, we followed a precise procedure to find the literature around well-established metrics and their meaning in industry practices (represented in Chapter 2, Table 2.1) Incorporating a few of those process metrics in order to assess process efficiency is the primary objective. Quality measurements are unquestionably crucial, but they were merely excluded because they were not the focus of this study. Other process metrics were excluded because there is no practical way to calculate them all in the scope of the current study. Instead, we focused on those metrics that are the most impactful and practical to measure.

3.1 Research Questions

With an objective to evaluate the usefulness of Agile and Lean software process metrics for OSS communities, and to understand the extent of coherence of the proposed method for Open Source projects portraying diverse characteristics, a qualitative and descriptive data analysis methodology is proposed in this study. To achieve this goal, two main research questions are explored:

RQ1: Can Agile/Lean metrics provide useful insights about the efficiency of Open Source Software Development communities?

RQ2: Is there a consistent interpretation of Agile/Lean metrics based on the distinct characteristics of Open Source Software projects?

3.2 Research Methodology

Figure 3.1 elucidates the data collection process employed in this study. A set of metrics used in both Agile and Lean software development along with various OSS projects and characteristics are found by conducting an analysis of relevant studies. In Stage 2, a representative sample of OSS projects and characteristics are filtered in order to perform an exploratory study. Similarly, the process metrics are also included based on the relevance to the OSS ecosystem. In Stage 3, metrics included in this study are parameterized by processing each of them against the pre-selected OSS projects as shown in Chapter 5.

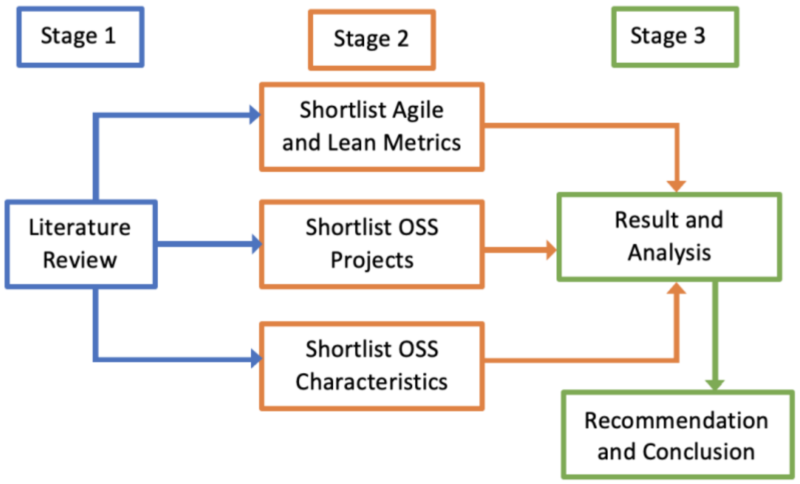

Stage 1
Stage 2
Stage 3
Shortlist Agile and Lean Metrics
Literature Review
Shortlist OSS Projects
Result and Analysis
Shortlist OSS Characteristics
Recommendation and Conclusion

Figure 3.1: Process of Data Collection

3.2.1 Representative Sample of OSS Projects

In order to represent stable software projects, we want to evaluate projects with a large enough data set, sufficient activity, and adequate history. Therefore, the methodology of the study starts with a search for Open Source Projects with a well-established, mature codebase maintained by a large development team. Project inclusion was based on the number of commits and a fixed boundary of no less than 20,000 commits is adopted. For discarding the selection of trivial communities, the selected projects are then filtered based on the number of contributors, defining the limit to at least 30 contributors in each project. The adopted projects are further refined to have at least 100,000 LOC; in this way, only large codebases are dealt with. After the initial inclusion criteria, we wound up with 40 OSS projects from GitHub and these are listed in appendix A.

With the intent to calculate the Agile process metrics accurately, retrieving the history of flow of work is necessary and hence the presence of a Scrum/Kanban board is critical. Therefore, in the next stage we tried to reduce the set of projects by filtering based on the presence of a project board. Among the list of 40 projects, 13 projects had a project board associated with them. Further refining is performed based on the age of the project. Given that every project undergoes a ramp-up phase, the early developmental trait of an OSS is different from when it matures. Compared to a mature project, the nature of work is different during the ramp up period. Therefore, projects that are 5 years or older are selected as evidence to prove that these projects have undergone several iterations in the Open Source ecosystem and have a set of differentiating characteristics. The chosen projects are thoroughly checked to ensure that they are all non-academic projects. Table

3.1. reports the characteristics of the subject projects in terms of a) their size measured as number of public releases issued, b) number of LOC, c) project start date and d) most recent stable release to prove that the project is still active.

Table 3.1: Characteristics of the Software Projects Considered in the Study, as Extracted from GitHub in April 2022

Project Name	**Project State Date**	**Major | Minor Releases**	**Stable Release**	**LOC**
Ansible	Feb 5, 2012	3 | 32	June 21, 2022	160K
Gatsby	May 17, 2015	5 | 103	February 8, 2022	400K
TensorFlow	Nov 1, 2015	3 | 35	May 16, 2022	3.09M
ASP.NET Core	Dec 8, 2013	6 | 10	November 8, 2021	1.47M
Flutter	Oct 19, 2014	4 | 58	July 1, 2022	1.85M
Salt	Feb 20, 2011	3 | 21	August 25, 2022	768K
React-Native	Jan 25, 2015	0 | 70	March 30, 2022	405K
Bootstrap	April 24, 2011	5 | 24	July 19, 2022	186K
Kubernetes	June 1, 2014	2 | 46	August 23, 2022	1.77M
Scikit-Learn	Jan 3, 2010	2 | 11	December 25, 2021	176K
Electron	March 10, 2013	22 | 47	August 25, 2022	132K
Netty	Aug 3, 2008	2 | 12	August 26, 2022	12.7M
PDF-JS	April 24, 2011	3 | 33	May 14, 2022	134K

13 projects were still too many, therefore additional screening of repositories is made based on the information gathered by carefully inspecting the project board, where the age of the project board is measured by the age of the cards it consists of. Project boards that are created after July 2021 are excluded from the list as they do not have sufficient information regarding the flow of work. Based on this exclusion criteria, PDF-JS, which was created in February 2022, is eliminated from the list of pre-selected projects. Similarly, repositories with less than 100 cards across all the projects are also removed due to limited data needed to measure the process metrics. Netty and Electron are hence removed as they consisted of 94 and 76 cards, respectively. Furthermore, any repository that does not have consistent creation of issue cards is also eliminated from the list. This results in the exclusion of React-Native, Bootstrap, Scikit-Learn and Kubernetes as there are no records of new cards created for at least a year since the beginning of the project board. TensorFlow was removed from the list because it only has one project board for tracking the status of pull requests (PR). This falls short of providing the entire flow work required to compute the process metrics. For the same reason, one of the Salt projects (PRs to port to master) is also disregarded when calculating the metrics.

It is essential to have adequate data with detailed information in order to measure the process metrics against the preselected projects. This requires a substantial number of cards on the project board used by numerous users thus highlighting the popularity of the candidate projects to be taken into consideration. To begin with, these projects are filtered based on the total number of Contributors, total number of Commits, number of Forks and number of Stars on a particular project. A fork is a replica of a repository and is frequently employed to either suggest changes to another person's project for which one does not have write access or to use another person's project as the basis for a new proposal. A repository can be forked such that a copy can be made, and necessary modifications are done without impacting the original repository. Star counts are routinely used by researchers as a proxy for project popularity [60]. In the study conducted by Munaiah et al. [61], the stargazers-based classifiers exhibit high precision of 97% in identifying popular projects. Table 3.2 provides a summary of the classifiers mentioned above as of October 1, 2022. A limit of 1K verified contributors, 20K number of commits, 5K Forks and 10K Stars is applied in the study.

Table 3.2: Proof of Popularity of Shortlisted Projects Based on Total Number of Contributors, Total Number of Commits, Number of Forks and Number of Stars

Project Name	**Total number of Contributors**	**Total number of Commits**	**Stars**	**Forks**
Ansible	5.3K	53K	54.7K	22.4K
Gatsby	3.9K	20.6K	53.6K	10.4K
ASP.NET Core	1.06K	50.3K	29.7K	8.3K
Flutter	1.07K	31.4K	145K	23.4K
Salt	2.3K	114.4K	12.8K	5.3K

Given that the current research involves the study of Process metrics related to the flow of work, it is important to ensure that these 6 selected projects are active. The criteria to verify that includes a) authentic commits in the recent past, b) releases generated in a timely manner, c) communication between users and developers in the form of comments, d) number of issues closed and e) number of merged pull requests. Table 3.3 and Table 3.4 provide a summary of the measures mentioned above as of October 1, 2022. This data confirms our assumption that the nominated projects are active at the time of the research.

Table 3.3: Proof of Activity of Shortlisted Projects Based on Commits from September 1, 2022, to October 1, 2022.

Project Name	**Number of Scrum/ Kanban boards**	**Number of overall cards**	**Commit Summary**
Ansible	14	1053	Excluding merges, 48 authors have pushed 161 commits.
Gatsby	4	143	Excluding merges, 37 authors have pushed 309 commits.
ASP.NET Core	6	527	Excluding merges, 42 authors have pushed 324 commits.
Flutter	173	4,284	Excluding merges, 94 authors have pushed 649 commits.
Salt	6	280	Excluding merges, 51 authors have pushed 296 commits.

Table 3.4: Proof of Activity of Shortlisted Projects Based on Number of Releases, Comments, Merged Pull Requests and Closed Issues.

Project Name	**Releases from 01-Jul-2022 to 01-Oct-2022**	**Pull Request Comments from 01-Sept-2022 to 01-Oct-2022**	**Merged Pull Requests during 01-Sept-2022 to 01-Oct-2022**	**Closed Issues during 01-Sept-2022 to 01-Oct-2022**
Ansible	11	199	160	124
Gatsby	7	99	110	34
ASP.NET Core	9	419	280	335
Flutter	5	101	642	1133
Salt	3	95	127	96

Finally, this results in a list of 5 projects namely Ansible, Gatsby, ASP.NET Core, Flutter and Salt are chosen for this study. Given that there are millions of OSS projects on

GitHub, the search was for stable, large projects with a good flow of work. It is important to remember that this is not an exhaustive list, merely a representative one.

Black Duck Open Hub [62] is a website which provides a web services suite and online community platform that aims to index the OSS development community. Comprehensive details and attributes of all the chosen projects used in this study based on the information available from OpenHub are shown in Appendix B.

3.2.2 Included/Excluded Agile and Lean Metrics

Various metrics have been identified in the studies where their benefits outweigh the overhead involved to track them. The emphasis is on research and experience reports that provide empirical evidence regarding actual use of metrics in Agile contexts. Metrics that are utilized solely for academic or comparative reasons, i.e., metrics which are not used to help software development activity, are excluded. Studying metrics, their applications, and their advantages in an Agile context is crucial since failing to grasp the context would restrict the understanding and use of the findings in many situations [63].

Thirty-five metrics were identified during our review of prevalent Agile/Lean process metrics (see Chapter 2). The method for identifying metrics of high influence is based on the number of instances a particular metric occurs in the original research. Out of those 32 metrics, 13 metrics namely, Lead Time, Cycle Time, Escaped Defects, Work in Progress (WIP), Velocity, Work Capacity, Focus Factor, Adopted Work, Found Work, Delivery on Time, Throughput/Story Rate per Iteration, Burndown Chart and Release Frequency appear to be recurring in more than one primary study. The purpose of this study is to determine whether OSS software practices may be improved and made more effective.

I wanted to test if the same measures that are used in Agile communities and Lean to promote efficiency would work with OSS.

With regard to the development of OSS projects, it is necessary to confirm the relevancy of these metrics. Referring back to the most prominent and prevalent process metrics in Agile/Lean Software Engineering as given in table 2.1 of chapter 2, the list of those included or excluded, along with their justifications, follows below.

1. Lead Time [31]:

Lead Time, in the context of Lean, is described as the period between the appearance of a new work in a workflow and its final departure from the system [38]. In other words, it is the amount of the time from when a customer makes a request to when it is delivered. With the use of this metric, benchmarks for the duration it takes for a task to complete from beginning to end can be established. With the use of this information, one can determine which development phases—including the time the work item spent in the Product Backlog—take the longest. The same reasoning is used to determine how much time has passed between when a work item was initially created on an OSS project and when it was completed. For lead time calculations, we use an interpretation of the GitHub project board, which is described in implementation chapter.

2. Flow Time [28]:

Flow Time is a key metric in identifying waste. In a traditional software development project, Flow Time has several advantages, including the use of it to schedule high priority features, to establish client demo dates and to evaluate if the release date will allow for the completion of all scheduled fixes. It is equally significant when referring to its relevance and applicability to OSS projects. Flow Time is the time spent on a work item from the point in time that the team has committed to completing the work. Kersten [28] uses DeGrandis' [64] understanding to create this interpretation of the concept of Lead Time. This metric is very important for us as it corresponds to the time when an Open Source team commits to proposed work, which in our dataset will correspond to when a project board card is created for an issue. We also compute “traditional” Lead Time and the time spent in the backlog but as Kersten points out in Open Source projects a huge volume of issues may be proposed that the team simply cannot groom all of those issues effectively.

3. Cycle Time [32]:

Cycle time in software development, whether it be traditional or Open Source, begins at the moment when the new arrival enters the “in progress” stage, and somebody is actually working on it [38]. Cycle time examines those parts of the process when teams are actively adding value to the current task, whereas lead time monitors the entire process since the arrival of work. It is a metric adopted from Lean principles and is one of the most essential KPIs for software development teams. Agile teams visualize the complexity or value of work using story points. Story points, however, do not communicate the complete picture as the team delays are not accounted for in the story points. Cycle time, which includes delays, is a measurement of how long it takes a team to finish a whole cycle. The complexity of the stories is irrelevant in this case. Teams can estimate what they can accomplish in a cycle or how quickly they can complete features by measuring their cycle times and using its average. Longer software development cycles result in ineffective development teams and delayed client deliveries.

4. Escaped Defects [32]:

Defects result in waste since they raise operating expenses without delivering value to the client [39]. Escaped defects is a simple metric that counts the defects for a given release that were found after the release date. These are defects not found by or escaped from the quality assurance team. Usually, end users identify these issues after being given access to the released version. Escaped defects are a crucial metric to track even in the context of OSS. However, the scope of our investigation is restricted to processing metrics that assess task flow. The future scope of work will therefore include quality measures like escaped defect.

5. Work in Progress (WIP) [35]:

WIP is the number of features or feature level integrations that a team is currently working on [38]. It provides a frame for the current workflow capability of the team. Larger WIPs are indications of significant inefficiencies in a team's workflow. One of the advantages of tracking WIP is that the bottlenecks in a team's delivery pipeline are immediately evident before a problem gets critical. WIP restrictions insist that working through the initial issue is preferable to beginning (but not finishing) new tasks. In other words, it discourages impeding the flow of work. WIP can also be referred to as Flow Load. Kersten's Flow Framework aggregates work items into four states: new, active, waiting, and done. This metric calculates the flow items in active or waiting states. In our work however, we aggregate work items into three states: To Do, In Progress, and Done. We do not have a way to identify work items in a waiting state, and in the context of an Open Source project, given the irregular nature of development team participation, it is both unimportant and impractical to expect waiting states to be identified.

6. Aging WIP [38]:

Aging WIP is another important indicator to consider in order to know how the team performed in similar circumstances in the past. Flow analysis is the foundation of Aging WIP management [38]. It enables users to see how work items are moving from the Requested column to the Done column on the Kanban board. In each of the sub-columns, work items spend varying amounts of time. To ascertain the cause and suggest improvements, it is critical to understand where the process is sluggish. For calculations of aging Work in Progress, we analyze the project board. This is provided in chapter 4 implementation.

7. Velocity [33]:

Velocity is calculated by measuring units of work completed in a given timeframe. Agile velocity measures how much work a single team completes in a sprint or iteration of software development. It can be represented as the slope in a typical burndown chart and indicates the number of story points accomplished over time. However, Flow Velocity [28] related to Agile velocity, is the number of flow items completed over a period of time. These are broken out by flow item type, but they do not have a weighted value such as story points. Related to our work, Flow Velocity is most closely related to throughput, though in our present research we have not filtered out by different work items (namely features and bugs).

8. Work Capacity [33]:

Work Capacity is the sum of all work reported during the Sprint, whether the Sprint Backlog Item toward which the work is applied finished or not. It is calculated based on the sum of actual work reported while Velocity is calculated based on the Original Estimates of work. Like Velocity, Work Capacity is calculated by estimating work. As work items are not estimated in a traditional way in OSS, it is challenging to acquire the data necessary to compute this metric.

9. Focus Factor [33]:

The number of deliverables that can be produced in an iteration is forecasted using the focus factor. The work capacity and velocity of a development team can be considered when calculating this number in an Agile environment. Focus Factor is excluded from the current analysis because it is a derived metric that is generated from the ratio of Velocity and Work Capacity.

10. Adopted Work [18]:

Adopted Work is any work that is moved from the Product Backlog to the Sprint as a result of the team meeting their original forecasted deadline earlier than expected. According to Downey and Sutherland [3] percentage of Adopted Work is calculated using the formulation: $\sum$ (Original Estimates of Adopted Work) $\div$ (Original Forecast for the Sprint). An OSS context for Adopted Work necessitates a little update where it is looked to see if a work item is transferred from the Product Backlog to the current project board during a release period. Even though there is not any formal planning prior to the start of a release, adopted work can be utilized as a sign of either the effectiveness or existence of informal planning.

11. Found Work [18]:

Found Work is additional, unanticipated work that is connected to a part of forecasted work that must be finished in order to deliver the original work item. The formula shown below is used by Downey and Sutherland [33] to calculate the percentage of found work: (Original Estimates of Found Work) / (Original Forecast for the Sprint). However, this metric is inapplicable in the context of an OSS project as the release serves as the sprint alternative. Releases, however, typically are not time-boxed like Sprints. Consequently, the initial work forecast is not officially documented. Simply said, as a result of this variation, there is no good enough approximation of found work to measure it at this time. This measure is yet another viable choice for future work.

12. Throughput [28, 18]:

Throughput is the number of tasks finished per time unit and represents the productivity level of the team in the past. In other words, the quantity of work items moving through a system or process in a certain amount of time is regarded as throughput. Considering a situation where it is necessary to know how many tasks a team can do each week, the delivered work items in the workflow must be calculated each day in order to obtain this data. If the team finishes two tasks on Monday, two on Tuesday, five on Wednesday, four on Thursday, and three on Friday, the formula for a team's average throughput rate is (2+2+5+4+3)/5 = 3.2 tasks. This suggests that the team can complete three jobs per day, or that this is the workflow's typical daily throughput rate. A team's past productivity is visually illustrated using the throughput histogram where the number of daily tasks that are completed and the time it took for those to occur are taken into consideration.

13. Burndown Chart [34]:

The project progress is displayed using a burndown chart (sprint and release) based on the overall effort and length of the remaining sprints. It represents the effectiveness of the team throughout the development process. The burndown chart also offers a summary of the project trend that may be used as a guide to estimate when the project will be finished. A burndown chart is a straightforward visual representation of how work advances. It identifies if the actual bulk of the work remaining (represented by the vertical bars) is behind (above) or ahead of (below) the optimum work remaining (the straight line) with only one glance. The accuracy of estimates has a direct impact on the value of burndown charts. Not only are time estimates frequently inaccurate, but estimation in an OSS project functions notably different from estimation in a typical Agile project. This necessitates the creation of a modified burndown chart that shows the flow of work items across a release timeframe. Given that OSS projects are believed to be feature-boxed rather than time-boxed, the ideal work remaining line estimate is generated using the tentative release window period. This updated burndown chart compares the progression of To-Do, WIP and Done work items over a release period in the form of a Cumulative Flow Diagram (CFD) [74]. A forecast can be obtained based on how work develops in OSS projects when this is implemented for a handful of significant releases in the past.

14. Delivery on Time [18]:

Delivery on Time is described as the ratio of features done in a planned release schedule. This metric measures whether the organization has been capable of delivering all features in a release which have been promised to customers, and what is the ratio of the promised content. In the context of OSS, it is the percentage of work items that are completed at the end of each release cycle when compared to the beginning of a release. Here, Delivery on Time is calculated as the percentage of completed work items at the end of each release cycle when compared to the beginning of a release.

15. Release Frequency [31]:

Frequent releases lower technical and market risks, and the client may evaluate a real product in smaller increments rather than only seeing transitory outcomes from progress reports. This gives indicative of an improvement in configuration management discipline and capabilities. Early and frequent product releases are crucial for OSS projects. For the creation of these releases, it is crucial to have a clear procedure in place. This is required to make sure that all legal concerns are resolved and that a release is of sufficient quality to be helpful to users. The distribution of releases is also an essential component of any Open Source project as it makes it easier for individuals to try out the project, which raises the possibility of finding new contributors. Given the community in Open Source is linked to product innovation and adoption and that the involvement of the community is ultimately what propels Open Source activities, this becomes a crucial indicator to monitor. For the sake of this study, this metric is used as a primary filter on the other metrics stated above to gather more specific information on a granular level.

Table 3.5: List of Shortlisted Metrics and How Each of Them are Computed.

Metric Name	**How to Compute**
Flow Time	Number of days spent on a work item from the point in time that the team has committed to completing the work
Lead Time	Number of days from when a customer makes a request to when it is delivered
Cycle time	Number of days between when work is started and when it is completed
Work in Progress (WIP)	Count of active items on the project board.
Aging WIP	Count of number of days work items sit in the active state.
Throughput	Number of work items completed for each day for the entire timeline of the project
Time on Product Backlog	Amount of time a work item typically spends on the Product Backlog before getting on a project board
Cumulative Flow Diagram	Progression of work items in To-Do, WIP and Done over a release period.

Table 3.5 provides a brief explanation of how these metrics are calculated. The implementation chapter (chapter 4) that follows explains how we interpret the data from the GitHub project board and the GitHub API to measure the metrics we've included in this study.

3.2.3 Included OSS Characteristics

Eric Raymond's [65] The Cathedral and the Bazaar model provides a solid foundation for understanding the various OSS development styles used. While traditional software engineering projects and OSS projects share some similarities, understanding their distinctive qualities is the main goal of this section. Some of the key differentiators are: (a) Time-boxed versus feature-boxed releases, (b) the project governance model and the structure of the developer community, (c) the regularity and trend of the release schedule, (d) the makeup of the target audience, and (e) monetization are some of the key differentiators.

Agile time-boxes encompass more than just splitting lengthy projects into smaller chunks of time. It supports product development in an iterative yet regulated way through a well-defined Agile approach. The purpose of time-boxing is to specify and establish a time restriction for each task. Scrum has a time-box for each of its five events. The duration of the Sprint is governed by time-boxing. The timeframe for the Sprint Planning meeting is set when a team starts. The time-box for Sprint Planning should be as brief as possible given the length of the Sprint. The Daily Scrum, a time-box of 15 minutes for every 24-hour period, aids the Scrum Team in synchronizing operations and highlighting any obstacles to reaching the Sprint Goal. Similarly, both Sprint Review and Sprint Retrospective are also time-boxed. By focusing on accomplishing the task or goal before the time restriction runs out, the time-box essentially promotes team productivity.

It is worthwhile to point out the uniqueness of functioning of the OSS community, where instead of time-boxes, the tasks are accomplished in a feature-boxed manner. Generally, with OSS, it is easy to become distracted with nice-to-haves or development tools instead of focusing on creating value. However, feature-boxing solves this problem as the team is committed to delivering high-quality software on time and within budget. The core team meets informally to develop a list of features that must be finished before the project’s next release. Additionally, feature-boxing adheres to the OSS principle of early and frequent release. Bug reporting or contribution of ideas are not considered as development during this phase. The OSS projects release their latest version after the predetermined set of features is complete.

Many OSS projects are developed by hundreds or even thousands of developers. An OSS project, however, cannot depend on such a sizable developer population. In contrast to traditional projects where each developer contributes the same amount of effort, the developer community in an OSS project can be classified into stable and transient developers [57]. A stable developer consistently provides code to the project, whereas a transient developer only makes isolated code fixes, both from trivial to considerable ones. One can instantly notice that most projects only have a small number of stable developers based on developer contribution. As a result, the developer community in OSS can be divided into three groups: i) a core team that initiates the project, manages the architectural style, and directs the development, ii) a larger group that uses the deliverables and recommends updates and enhancements and iii) an extended group that simply uses the deliverables. With the aid of this classification, it is possible to handle important aspects like the existence and quantity of a core team and the role of developers in the project.

Numerous details about an Open Source project are revealed by the releases' variability. If the project's rate of evolution is measured in terms of new releases per period, one can deduce that aged projects that do not evolve are likely to have been abandoned, OSS projects are required to be released regularly because of this, which also reveals information about the project's health. The pattern of release distribution can reveal key facts, including the interval between every version and the significant software development activities completed in-between each release.

The intended audience for OSS projects is another attribute that needs to be discussed. With over 25 million other Open Source projects, an OSS project must compete for both support and attention in order to succeed. That fact alone implies that marketing is essential for the Open Source initiative to succeed. The target audience will differ depending on the project, but a typical target audience will include both the project's end users and other community members. The goal is for the community to embrace the concepts, demonstrate the same enthusiasm, and participate in the project. It is necessary to draw contributors to the project because they are the foundation of the Open Source community and are essential to its growth. Some of these contributors may be freelancers who work alone in their spare time, while others may be employees of an enterprise or software firms. Some may be free to work on any project they choose, while others may have been tasked by their employer with contributing to a particular project. Additionally, organizations engage developers to collaborate on specific OSS projects so that they can be incorporated in other projects the company is working on.

Understanding OSS's monetization model is especially crucial considering that it is free and open. Businesses may use Open Source in a variety of ways while developing their business models. There are multiple ways an OSS project could be monetized. Large OSS projects are often owned by a non-profit organization. In the case of Mozilla, the Mozilla Corporation receives royalties from the contracts created using the open-licensed software. Some OSS projects have Open SaaS model where they offer a paid Software-as-a-Service alternative built on Open Source code. The OSS project is also funded using the multi-licensing idea. Table 3.6 provides a list of distinct characteristics of OSS projects when compared to commercially developed software projects.

Table 3.6: Distinct Characteristics of OSS Compared to Commercially Developed Software

Characteristic	**OSS Projects**	**Commercially developed software**
Software development and delivery approach	Feature-boxed	Time-boxed
Target audience	End users and other community members	Customers
Governance model	Stable/Core developers versus Transient developers	Commercial developers
Project's health depends on the regularity of releases	Yes	No
Monetization model	Rely on royalties from the contracts, Software-as-a-Service model, multi-licensing	Software Licensing or sales

In summary, this chapter set the context for this research by providing details about the research questions, research method for including the software process metrics along with OSS projects and its characteristics. The next chapter will talk about the design process and the implementation of the applying Agile/Lean software process metrics to the included projects.

CHAPTER 4

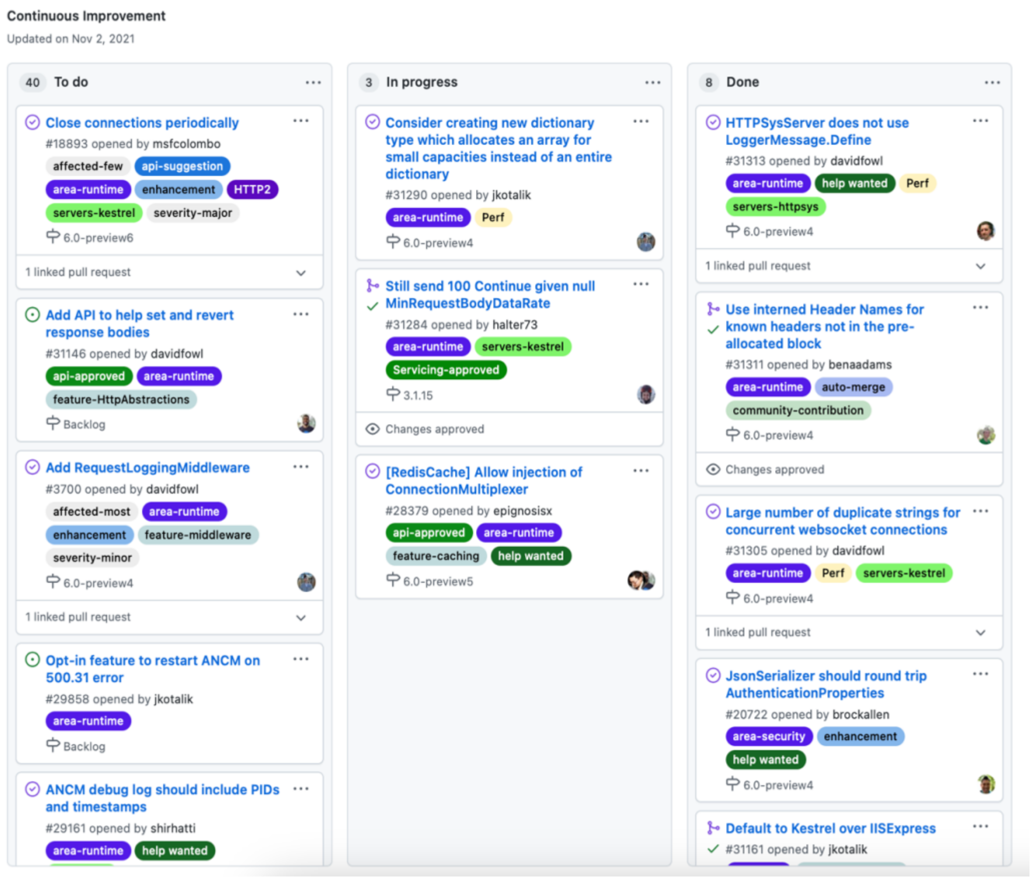
DESIGN AND IMPLEMENTATION

The primary focus of the implementation is to integrate the data with a report generator that helps in visualizing the utility of the Agile/Lean software process metrics as discussed in Chapter 3. The reader should be aware that this thesis did not primarily focus on creating an algorithm to retrieve data from GitHub. Python scripts are created to access the GitHub APIs to fetch all the essential data from the repository. This chapter describes the report generation tool's implementation process and outlines its key features.

4.1 Retrieve Information of Kanban Project Board from GitHub REST APIs

The Open Source projects dataset is compiled from GitHub, a social programming environment that provides source code management and collaboration capabilities for each project, including bug tracking, feature requests, task management, and wiki. GitHub boasts of having one of the largest developer communities in the world. As of January 2020, there are more than 190 million repositories and over 40 million users [66].

GitHub projects is an adaptable, flexible tool for planning and tracking work on GitHub [67]. A project is a flexible spreadsheet that connects with GitHub pull requests and issues to help in properly organizing and tracking the work. By filtering, sorting, and grouping the issues and pull requests, adding custom fields to maintain metadata unique to the team, and visualizing work with configurable charts, one can create and configure numerous perspectives. Instead of imposing a certain methodology, a project offers flexible features you can tailor to your team's goals and procedures. Figure 4.1 gives a visual representation of one of the projects in ASP.NET Core for reference.


Continuous Improvement
Updated on Nov 2, 2021

40 To do

Close connections periodically
#18893 opened by msfcolombo
affected-few api-suggestion area-runtime enhancement HTTP2 servers-kestrel severity-major
6.0-preview6
1 linked pull request

Add API to help set and revert response bodies
#31146 opened by davidfowl
api-approved area-runtime feature-HttpAbstractions
Backlog

Add RequestLoggingMiddleware
#3700 opened by davidfowl
affected-most area-runtime enhancement feature-middleware severity-minor
6.0-preview4
1 linked pull request

Opt-in feature to restart ANCM on 500.31 error
#29858 opened by jkotalik
area-runtime
Backlog

ANCM debug log should include PIDs and timestamps
#29161 opened by shirhatti
area-runtime help wanted

3 In progress

Consider creating new dictionary type which allocates an array for small capacities instead of an entire dictionary
#31290 opened by jkotalik
area-runtime Perf
6.0-preview4

Still send 100 Continue given null MinRequestBodyDataRate
#31284 opened by halter73
area-runtime servers-kestrel Servicing-approved
3.1.15
Changes approved

[RedisCache] Allow injection of ConnectionMultiplexer
#28379 opened by epignosisx
api-approved area-runtime feature-caching help wanted
6.0-preview5

8 Done

HTTPSysServer does not use LoggerMessage.Define
#31313 opened by davidfowl
area-runtime help wanted Perf servers-httpsys
6.0-preview4
1 linked pull request

Use interned Header Names for known headers not in the pre-allocated block
#31311 opened by benaadams
area-runtime auto-merge community-contribution
6.0-preview4
Changes approved

Large number of duplicate strings for concurrent websocket connections
#31305 opened by davidfowl
area-runtime Perf servers-kestrel
6.0-preview4
1 linked pull request

JsonSerializer should round trip AuthenticationProperties
#20722 opened by brockallen
area-security enhancement help wanted
6.0-preview4

Default to Kestrel over IISExpress
#31161 opened by jkotalik

Figure 4.1: ASP.NET Core’s Continuous Improvement Project Board

Appendix C has all the specifics on how data was obtained from the GitHub APIs using a customized scraper that was created in Python.

4.2 Calculating Agile/Lean Process Metrics:

Here is a detailed explanation of how these metrics were calculated using the GitHub APIs, scraper, and a database. The focus of this study is on the data that was created from 2018, therefore it is necessary to note that all events performed on issues prior to 2017 are not documented. Every card on the project board corresponds to a single issue, which may be a pull request or a conventional issue. According to GitHub, every pull request is an issue, but not every issue is a pull request. Additionally, it is true that one issue may be associated with multiple cards. As a result, if an issue is added to multiple projects, the project in which the card is completed first, along with its associated start date, is taken into account since it signifies the first instance when the work was completed.

The date a card is added to a project is considered as the date the work is created, with the state being to-do. The work started date is when a card is moved to the in-progress state or to a state that indicates a similar work. Likewise, the date a card moves to the done state is taken into account as the date of work completion.

A card is believed to have completed two different items of work if it is moved from the to-do state to the done state and returned to the to-do state. The work was marked as completed when the card initially moved to the done state. The second iteration of a work is considered to be new work or rework. Like it does on an Agile project, rework can occur in one of two ways. One, requirements change after the work is completed, or two, software development teams discover better ways to design the system. But this is not applicable if a card is unintentionally moved to done. This can be detected by observing the time a card remains in the in-progress state. This is the reason it is important to have a ready-to-test column to which a card should be moved to as it signifies that the work is tested, and peer reviewed prior to being considered completed.

When a card is added straight to the in-progress state, the to-do state is also updated with this date resulting in both work created, and work started to have the same date. As a result of this, the Cycle Time and Lead Time for that specific issue will be equal. Similarly, if an issue is added directly to the done state, the same date is considered as when the work was added to the project.

Another consideration to note is when a card is placed in the deferred or suspended state, it is presumed that it was not finished, and there is no work connected with such a card. Such entries are not included in the study since they do not reflect what has been accomplished. This suggests that the study will only take into account the cards that are either finished, ongoing, or still active in the to-do state.

The fact that more than 25% of the cards were moved directly from to-do to done is one of the difficulties encountered when computing the Cycle Time metric. These cards were never moved to the in-progress state. The Cycle Time for such cards will be identical to the Lead Time. This variation should be taken into consideration as it could be the reason for why OSS communities are unable to precisely track the flow of work.

In this study work items are aggregated into three states: To Do, In Progress, and Done. These states important for metrics calculation. The implementation of all the metrics included in Chapter 3 is as follows.

1. Flow Time: The Lead Time in the current investigation is computed using the flow of work in the project board. It is assumed that the work is created when it enters the project board in the framework of the current study. The start time is the date that a specific card is newly added to a column in the board. This is determined using the help of GitHub events. The flow of work within the project board can be tracked with the help of GitHub events. When a card is added to a project board, an event type called ‘added to project’ is recorded. The end date for said card is the date at which the card is moved to the ‘Done’ column indicating that the work is

completed and ready to be shipped. An event called ‘moved columns in project" is used to track card movement inside project boards. The difference between the end date and the start date is referred to as the Lead Time.

2. Lead Time: Lead Time is the interval between the time a work item entered the project (rather than the project board). With the use of this metric, benchmarks for the duration it takes for a task to complete from beginning to end can be established. With the use of this information, one can determine which development phases—including the time the work item spent in the Product Backlog—take the longest. The same reasoning is used to determine how much time has passed between when a work item was initially created on an OSS project and when it was added to the project board.
3. Cycle time: The time the card was moved to the in-progress state is used to calculate the Cycle Time. The start time is the time when a particular card is either moved to or freshly added to the in-progress column on the board. This indicates that the work on a particular task has begun. The date when a card is moved to the ‘Done’ column, indicating that the job is finished and ready to be dispatched, is the end date for that card. The Cycle Time is the interval between the End Date and the Start Date.
4. Work in Progress: The number of work items that are in progress or in any other similar active states on the project board is used to calculate the WIP in this study.
5. Aging WIP: Aging WIP is calculated by measuring the number of days work items sit in any of the active states once the work has started.

6. Throughput: In this study, Throughput is calculated by counting the number of work items completed over a release period. It indicates the total value-added work output by the team over a period of time.
7. Cumulative Flow Diagram: is the progression of work items in To-Do, WIP and Done over a release period.
8. Time on Product Backlog: Amount of time a work item typically spends on the Product Backlog before getting on a project board
9. Adopted Work: Number of cards were created during a release period

4.3 Use PostgreSQL to Store the Data Fetched from GitHub APIs

PostgreSQL, commonly known as Postgres, is a relational database management system that is free and Open Source, with an emphasis on extensibility and SQL conformance. One can securely store complex and enormous volumes of information using Postgres. It aids programmers in creating crucial settings, managing administrative activities, and creating even the most complex programs. PostgreSQL, MariaDB, and CUBRID are just a few examples of free Open Source RDBMS. PostgreSQL supports the ANSI: SQL2011 database standard and most of its features, including transactions. Due to separation of client and server in PostgreSQL, any changes in the database engine does not affect the client. Using an RDBMS like PostgreSQL produces better performance for a precise and structured data model. [68].

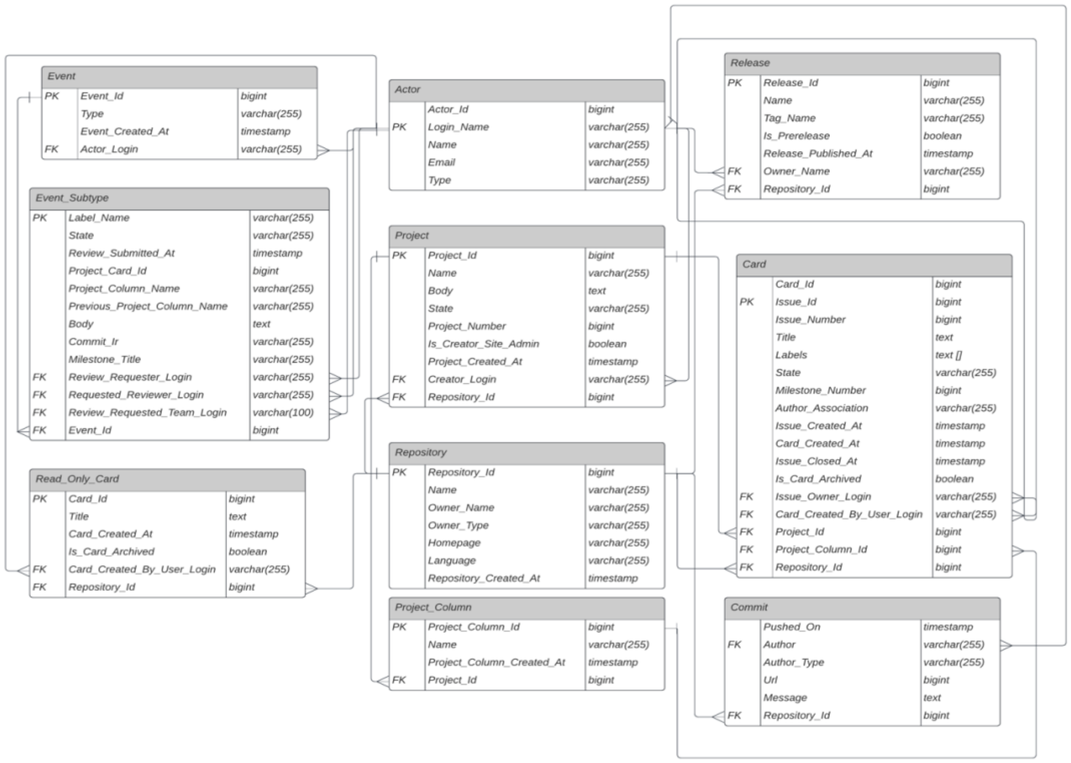


Figure 4.2.2: Database ER Diagram

SQL is a declarative programming language used to create and operate data in a relational database. Typically, SQL uses data tables to store information. These tables are interconnected and have a predefined data template. The schemas, records, logs, and constraints of the table are all contained in the PostgreSQL database. Because databases are rigorously segregated, the user is unable to access two databases simultaneously. DML (Data Manipulation Language) commands are used to manipulate the data in the PostgreSQL database. The PostgreSQL Schema specifies the general layout of the database's logical data structure, which includes all the necessary tables, data types, indexes, functions, stored procedures, and other components. For each user accessing the program, a separate Schema is defined in the database, preventing conflicts and unneeded interference. The database ER Diagram for our implementation is given in Figure 4.2.

Information regarding repositories, projects, columns in each project, cards in each column, the issue each card is associated with, all the events performed on these issues along with the data related to overall releases and commits in the repository is stored in the Postgres database for this study.

4.4 Establishing a Connection Between the Postgres Database and Metabase to Produce Visualizations

The Postgres database is then connected to Metabase, an Open Source business intelligence application that allows users to create charts and dashboards using data from a variety of databases and data sources. Data visualization tools such as Metabase offer a straightforward way to predict and understand trends and patterns in data by employing visual elements like graphs, charts, and maps. They assist in categorizing and organizing data based on subjects and themes, which makes it simpler to interpret.

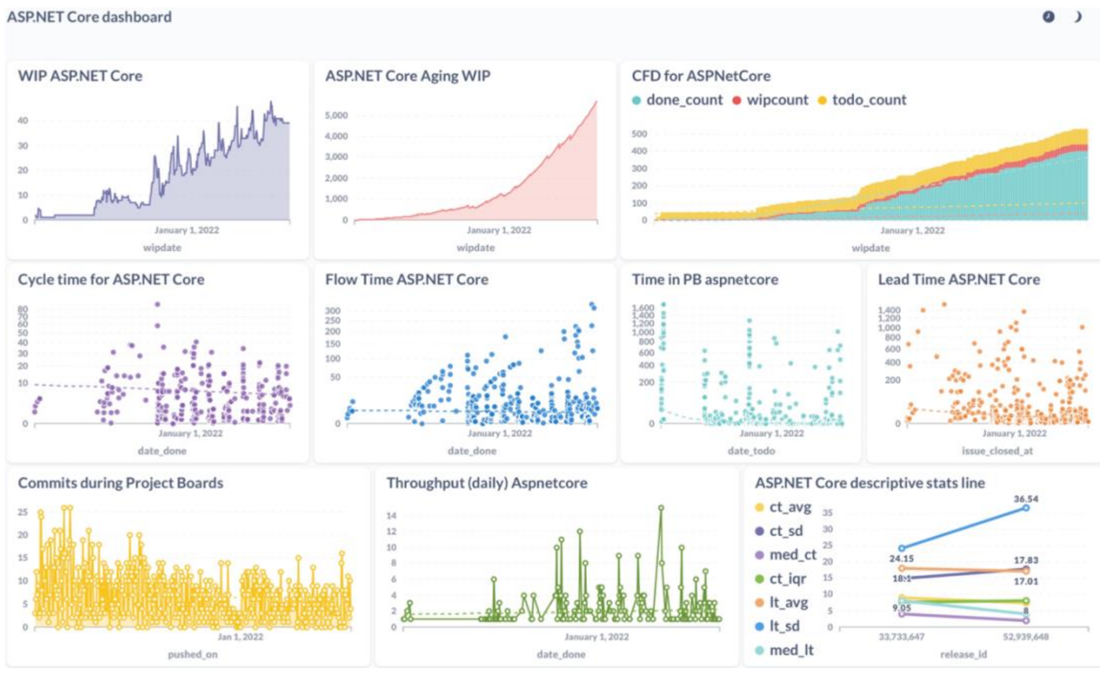


Figure 4.4.1: Dashboard for ASP.NET Core

My database (discussed above) was uploaded to Metabase. SQL queries were written based on this. Many visualizations generated using Metabase are presented below in Chapter 5. Dashboard for ASP.NET Core is given in Figure 4.4.1. Exploratory questions will be formed around metrics outcomes based on the observations made using these dashboards.

CHAPTER 5

RESULTS AND ANALYSIS

The process for applying Agile/Lean metrics to the OSS projects included in this study is described in Chapter 4. In the upcoming section 5.1, exploratory questions will be formed around metrics outcomes and for completeness additional questions have been pursued that are presented in section 5.2.

5.1 Inductive Reasoning

As exploratory research, my approach is to gather observations about the data by examining visualizations of flow metrics and coalesce these observations to form exploratory questions. From there, detailed data-driven queries drill down to root causes behind these phenomena. This inductive research process demonstrates the efficacy of applying Agile/Lean process flow metrics to OSS development processes. In this chapter, metric outcomes are examined, and observations made that prompt exploratory questions about the behavior to these projects. These questions will help in determining whether Agile/Lean metrics offer insight into the project's flow efficiency. For some of the explorations, additional drill down investigations is conducted to identify the root causes of such behavior. Table 5.1 provides a guide for the exploratory questions that were formulated during this analysis, and the narratives below present the drill-down investigations and conclusions drawn from them.

Table 5.1: Exploratory Questions and Observations

Exploratory Question	**Metrics**	**Observations**
EQ1. Is flow of work periodic in most OSS projects?	Flow Time, Cycle Time, Throughput, WIP, Aging WIP, Lead Time	Three of the five OSS projects we looked at appear to have a flow-of-work cadence based on the periodic pattern we can visually observe for Cycle Time and Flow Time.
EQ2. Can the structure of the development team and the governance model influence the regularity of work?	Throughput, WIP, Flow Time	ASP.NET Core has a higher consistent cadency of flow of work than other projects
EQ3. Do OSS communities have a concept of Adopted Work?	Flow Time, Cycle Time, Throughput, Aging WIP	Team is investing time into closing issues that have been around for a significant amount of time, particularly after clusters of shorter-term work
EQ4. Do OSS project boards on GitHub represent the flow of work accurately? Or do they just use the project board as records of past activity?	Flow Time, Cycle Time, WIP	There are various issues with Cycle Time and Flow Time are almost zero, suggesting the board is not representing reality.
EQ5. Do OSS projects' left-skewed distribution patterns reveal anything significant about the work being done?	Flow Time, Throughput	The spread of cards we see visually in Flow Time and Cycle Time suggests that 4 of the 5 projects we examined has a left-skew distribution pattern
EQ6. Can visualizations such as Cumulative Flow Diagrams help OSS project teams to find out bottlenecks in their process?	Cycle Time, Throughput, WIP	Inconsistencies such as drastic growth in the work items added to the To Do state that does not correlate with the number of work items completed both in Salt and Flutter's flow of work
EQ7. What can be inferred about the success of OSS project's release lifecycle from work completed schedule?	Flow Time, Cycle Time, Throughput, WIP, Aging WIP	Several issues were resolved during a small window of time

OSS, GitHub, and Agile/Lean have particular terminology that we map between these spaces detailed in Chapter 4 and summarized here for understanding. Issues are a list of all the modifications to the product while a card is composed of one issue that is on the project board. As described in the Scrum Guide [53], a Product Backlog is an evolving, ordered list of everything that is needed to improve the product and is the single source of requirements for any changes to be made [69] and a Sprint Backlog is a subset of the Product Backlog items selected with an actionable plan for delivering the increment. In the context of this study Product Backlog is the Issues tab on GitHub and Sprint Backlog are the cards associated with an issue that are part of the project board.

In previous chapters the definition of included metrics (Chapter 3), its interpretation in relation to OSS, and their calculations (Chapter 4) are covered. Next, I present and discuss the Exploratory Questions (EQs). In the first 3 cases I also present drill-down analyses and conclusions to demonstrate how Agile/Lean metrics may be impactful for OSS projects and communities, and also situations where these metrics may need re-interpretation or perhaps are not as meaningful compared to commercial software development processes.

EQ1: Is the flow of work periodic in most OSS projects?

This EQ presented itself through the visual patterns of flow-related metrics.

Observation: The scatter plot representation of Flow Time and Cycle Time for Ansible, ASP.NET Core and Flutter is shown in Figure 5.1. Three of the five OSS projects we looked at appear to have a flow-of-work cadence based on the periodic pattern we can visually observe for Cycle time and Flow time.

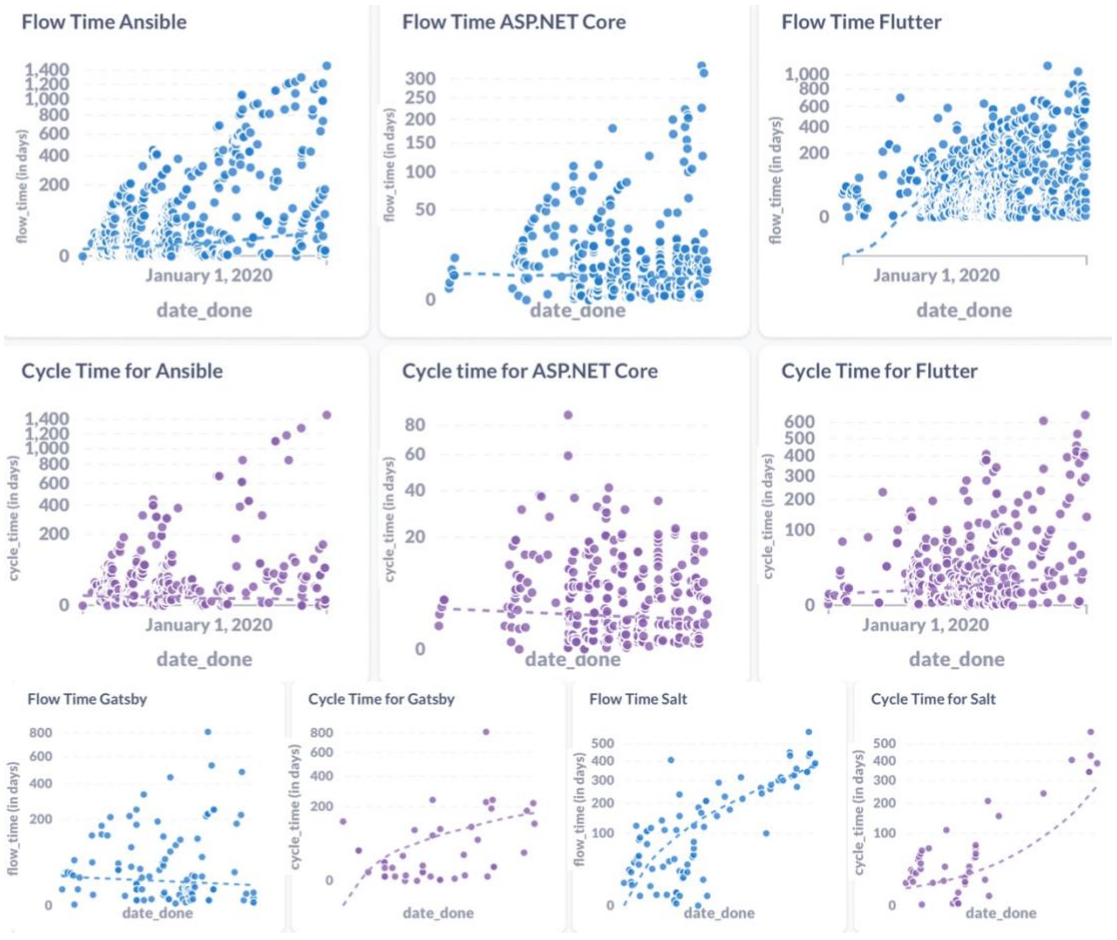


Figure 5.1: Flow Time and Cycle Time for the Included OSS Projects

Drill Down: Further investigation involves looking at Work in Progress and Throughput to check if there are similar collections of work done periodically. Figure 5.2 provides the representation for WIP and Throughput for Ansible, ASP.NET Core and Flutter.

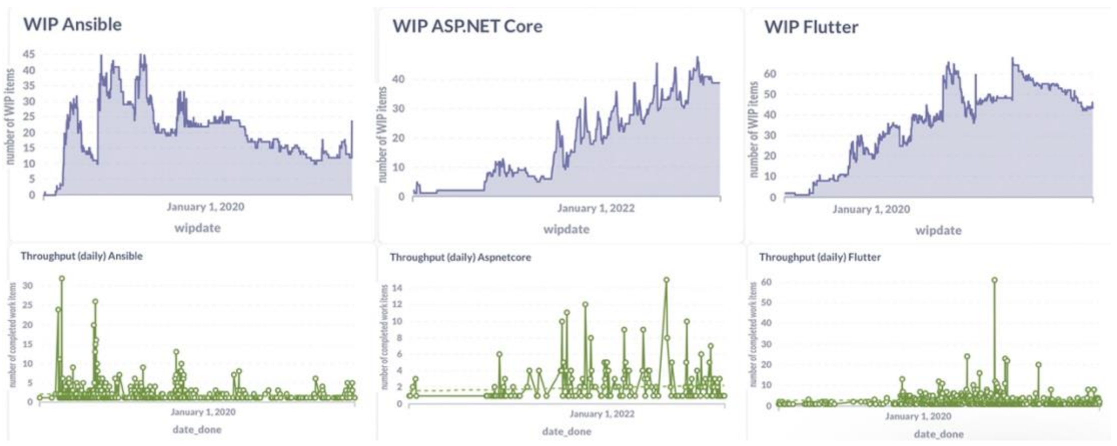


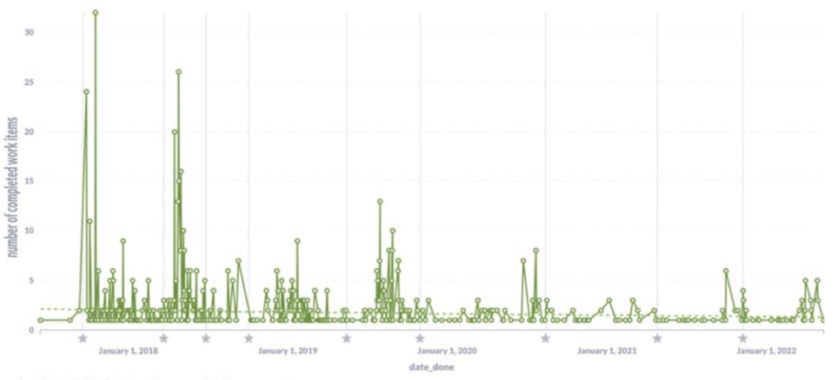
Figure 5.2: WIP and Throughput for Ansible, ASP.NET Core and Flutter.

After examination, it is apparent that there are spikes of completed work at periodic levels in all these projects. ASP.NET Core shows a regular periodicity while others have values occurring in abrupt bursts. There is no predictable periodicity in this intermittent nature. To further investigate the hypothesis of periodic cadence in these projects, release timelines are plotted on these charts. The Release Classification and Release Cadence of all the included OSS projects is provided in Table 5.2.

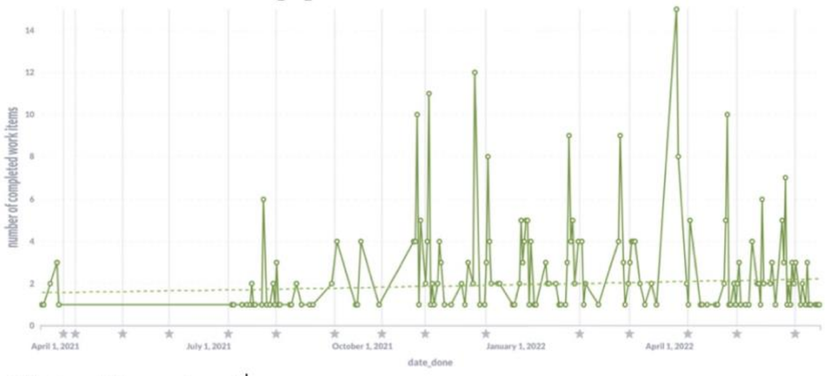
Table 5.2: Release Classification and Cadence of All the Included OSS Projects

Project	**Release Classification**	**Release Cadence**
Ansible	1.Major Release 2. Minor Release 3. Release Candidate	Team typically releases two major versions of the community package per year
ASP.NET Core	1. Long Term Support (LTS) 2. Current 3. Preview	New major .NET versions are released annually in November
Gatsby	1. Major 2. Minor	Typically releases 1 major version per year and 2 minor versions per month
Salt	1. Major 2. Feature	Goal is to cut a new feature release every three to four months.
Flutter	1. Stable Releases 2. Beta Release	Flutter ships updates on a roughly quarterly cadence

Ansible Throughput:



ASP.NET Core Throughput:



Flutter Throughput:

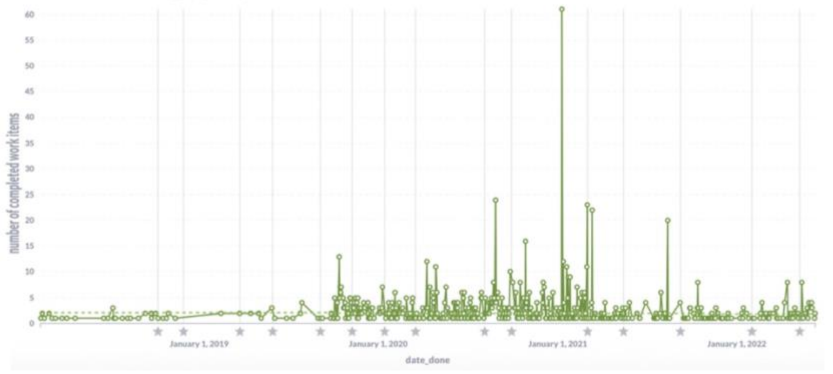


Figure 5.3: Major and Minor Releases Marked for Throughput in Ansible, ASP.NET Core and Flutter

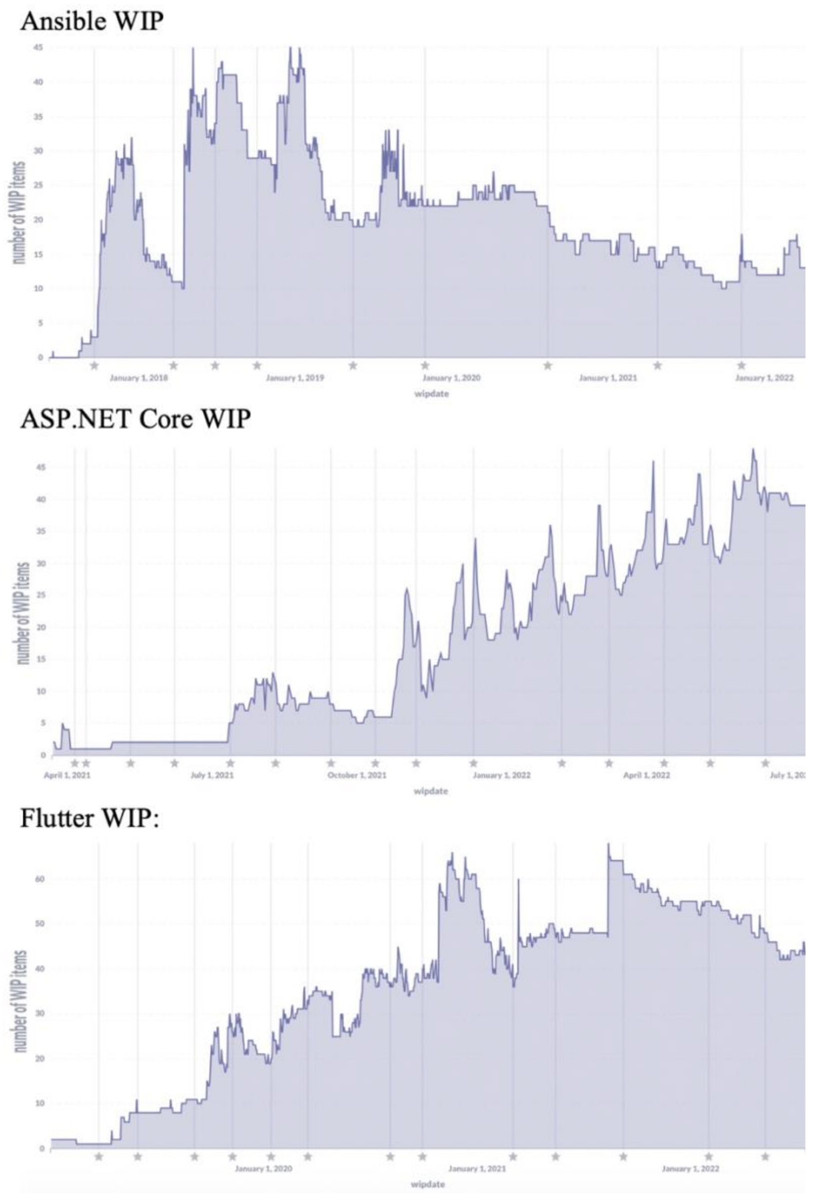


Figure 5.4: Major and Minor Releases Marked for WIP in Ansible, ASP.NET Core and Flutter

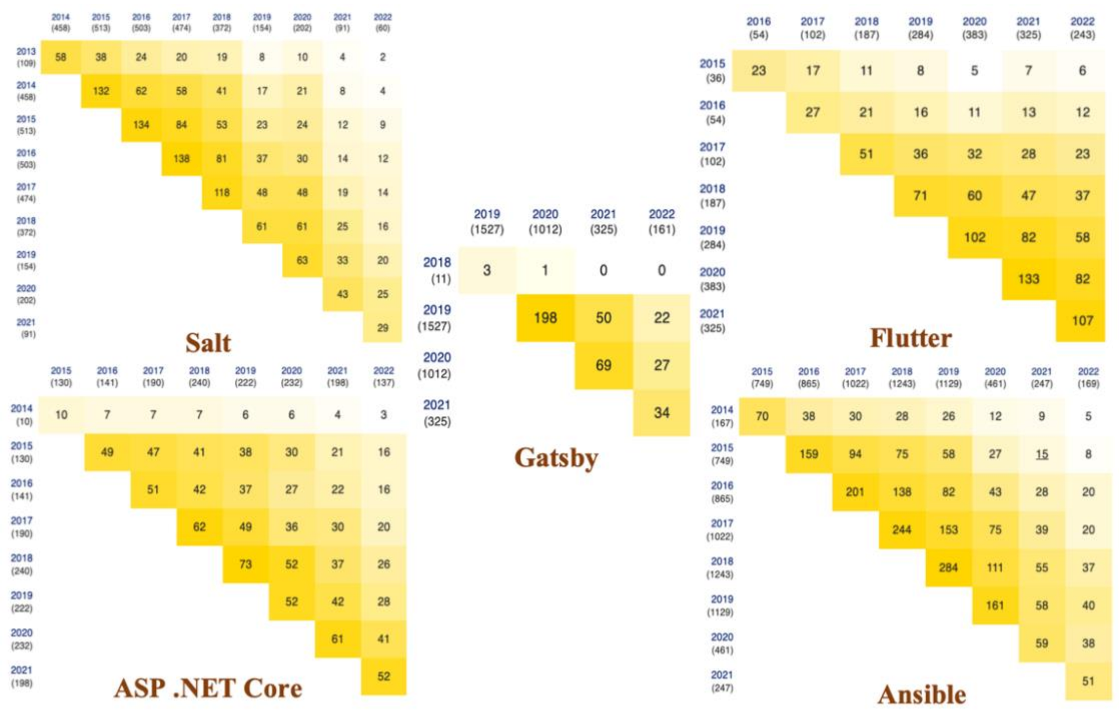
Figure 5.3 and Figure 5.4 shows the major and minor releases (gray vertical lines) marked for Throughput and WIP. It is quite evident that ASP.NET Core has a consistent release cadence, in contrast to the other projects, which do not provide a clear proof of consistency. According to ASP.NET Core, there is no regular or pre-defined schedule for minor releases. However, visualizations like these aids in our understanding of project workflows. When minor releases are highlighted on the Throughput chart, it demonstrates that the project has a periodic pattern. The number of WIP items increases over the course of the release before drastically decreasing at its completion. Nearly all release cycles exhibit this pattern. The continuous rise of WIP count within a release could be an indication that an informal release planning meeting was held during the beginning of a release. The periodic increase in the number of work items completed per day in the Throughput chart corresponds to the data in Flow Time and Cycle Time. For the majority of the releases, a significant percentage of work is finished just before the release's conclusion. If there is some consistency in the flow of work, the metrics we included can provide valuable feedback, and ASP.NET Core is an ideal illustration of that.

<u>EQ 2:</u> Can the structure of the development team and the governance model influence the regularity of work?

This EQ was established as a result of future investigations that was prompted by the study in EQ1.

Observation: The analysis of Drill Down from EQ1 demonstrates that ASP.NET Core has a higher consistent cadency of flow of work than other projects. This raises the question of whether the development team and governance model impact the flow of efficiency.

Drill Down: One of the ways of answering this question is by investigating the developer turnover. Developer turnover is a measurement of the number of developers (including part-time and volunteer, stable and transient contributors) who stop contributing to the project over a given period. Developer turnover is crucial for commercial software because it has financial repercussions, particularly in terms of the costs associated with hiring new personnel. In the context of OSS, large OSS projects typically have a distributed development team. According to studies, the success of a project is significantly impacted by developer turnover [70]. Frequent developer turnover may result in lost productivity due to a lack of relevant expertise and extra time spent understanding how projects work. As a result, emphasis has been placed on determining aspects associated with developer retention such as early participation, team collaboration and code over documentation.



Salt

	2014 (458)	2015 (513)	2016 (503)	2017 (474)	2018 (372)	2019 (154)	2020 (202)	2021 (91)	2022 (60)
2013 (109)	58	38	24	20	19	8	10	4	2
2014 (458)		132	62	58	41	17	21	8	4
2015 (513)			134	84	53	23	24	12	9
2016 (503)				138	81	37	30	14	12
2017 (474)					118	48	48	19	14
2018 (372)						61	61	25	16
2019 (154)							63	33	20
2020 (202)								43	25
2021 (91)									29

ASP .NET Core

	2015 (130)	2016 (141)	2017 (190)	2018 (240)	2019 (222)	2020 (232)	2021 (198)	2022 (137)
2014 (10)	10	7	7	7	6	6	4	3
2015 (130)		49	47	41	38	30	21	16
2016 (141)			51	42	37	27	22	16
2017 (190)				62	49	36	30	20
2018 (240)					73	52	37	26
2019 (222)						52	42	28
2020 (232)							61	41
2021 (198)								52

Gatsby

	2019 (1527)	2020 (1012)	2021 (325)	2022 (161)
2018 (11)	3	1	0	0
2019 (1527)		198	50	22
2020 (1012)			69	27
2021 (325)				34

Flutter

	2016 (54)	2017 (102)	2018 (187)	2019 (284)	2020 (383)	2021 (325)	2022 (243)
2015 (36)	23	17	11	8	5	7	6
2016 (54)		27	21	16	11	13	12
2017 (102)			51	36	32	28	23
2018 (187)				71	60	47	37
2019 (284)					102	82	58
2020 (383)						133	82
2021 (325)							107

Ansible

	2015 (749)	2016 (865)	2017 (1022)	2018 (1243)	2019 (1129)	2020 (461)	2021 (247)	2022 (169)
2014 (167)	70	38	30	28	26	12	9	5
2015 (749)		159	94	75	58	27	15	8
2016 (865)			201	138	82	43	28	20
2017 (1022)				244	153	75	39	20
2018 (1243)					284	111	55	37
2019 (1129)						161	58	40
2020 (461)							59	38
2021 (247)								51

Figure 5.5: Developer Count Retained Over Years

The developer count retained for the 5 OSS projects included in this study is given in Figure 5.5. The headers of each table represent the calendar year along with the number of overall contributors to the community. The cells represent the number of developers common between two calendar years. Developer Turnover Rate (Table 5.3) is calculated by comparing the last cell of each column using the formula: Developer Turnover rate = [(# of developers who left) / (total # developers that year)] x 100 [71].

Developer Turnover rate is important with respect to people leaving the community, but at the same time many of these communities are actually scaling up. This creates a two-pronged problem – there is a loss of experienced people while trying to scale up with new people. This can lead to a host of maturity problems for organizations and their processes and products, such as architectural drift, knowledge loss, and especially in OSS projects, a loss of focus or direction.

Table 5.3: Developer Turnover Rate

OSS Project	**2018**	**2019**	**2020**	**2021**
Gatsby	72.72%	87.03%	93.18%	89.53%
Salt	83.6%	59.09%	78.71%	68.13%
ASP.NET Core	69.58%	76.57%	73.70%	73.73%
Ansible	77.15%	85.73%	87.20%	79.35%
Flutter	62.03%	64.08%	65.27%	67.07%

Table 5.4 shows the increase or decrease in the number of contributors who made at least one code commit to the open source project.

Table 5.4: Overall Change in the Contributors to the Community

OSS Project	**2018**	**2019**	**2020**	**2021**	**2022**
Gatsby	X	X	-33.72%	-67.88%	-50.46%
Salt	-21.51%	-58.6%	31.16%	-54.95%	-12.08%
ASP.NET Core	26.31%	-7.5%	4.5%	-14.65%	-30.8%
Ansible	21.62%	-9.17%	-59.16%	-46.42%	-31.57%
Flutter	83.33%	51.87%	34.85%	-15.14%	-25.23%

Table 5.5: Percentage of Developers with One or More Years of Experience

OSS Project	**2018**	**2019**	**2020**	**2021**	**2022**
Gatsby	X	0.19%	19.56%	21.23%	21.11%
Salt	31.72%	39.61%	31.18%	47.25%	48.33%
ASP.NET Core	25.83%	32.88%	22.41%	30.80%	37.95%
Ansible	19.62%	25.15%	34.92%	23.88%	30.17%
Flutter	27.27%	25%	26.63%	40.92%	44.03%

Table 5.5 is the number of developers with one or more years of experience and can be calculated for any number of years of experience since the inception of the project. This data makes it quite evident that Flutter has the lowest developer turnover rate, with ASP.NET Core coming in second. Therefore, in response to the EQ 2, this may be one of the factors that influenced the regularity of work in ASP.NET Core.

Drill Down: This inquiry can also be answered by examining the core developers of the project. Microsoft offers support for ASP.NET Core. How many core contributors are Microsoft employees? Based on GitHub commits, Figure 5.6 shows the developers who have made the most contributions to the project.

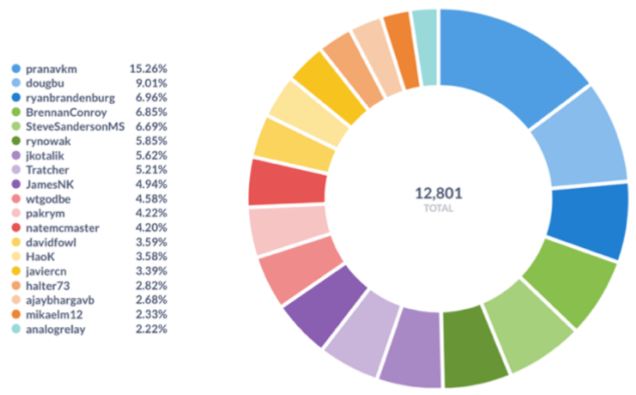


Figure 5.6: Developers with Highest Contributions to ASP.NET Core.

The following inquiry is whether these contributors are for-profit developers or volunteers. Investigation reveals that all 20 of the top developers listed here work for Microsoft. So ASP.NET Core is a perfect example of OSS project where the governance model is skewed towards industry practitioners paid on their job to develop and maintain this project. This explains why the project may have a regular, consistent behavior.

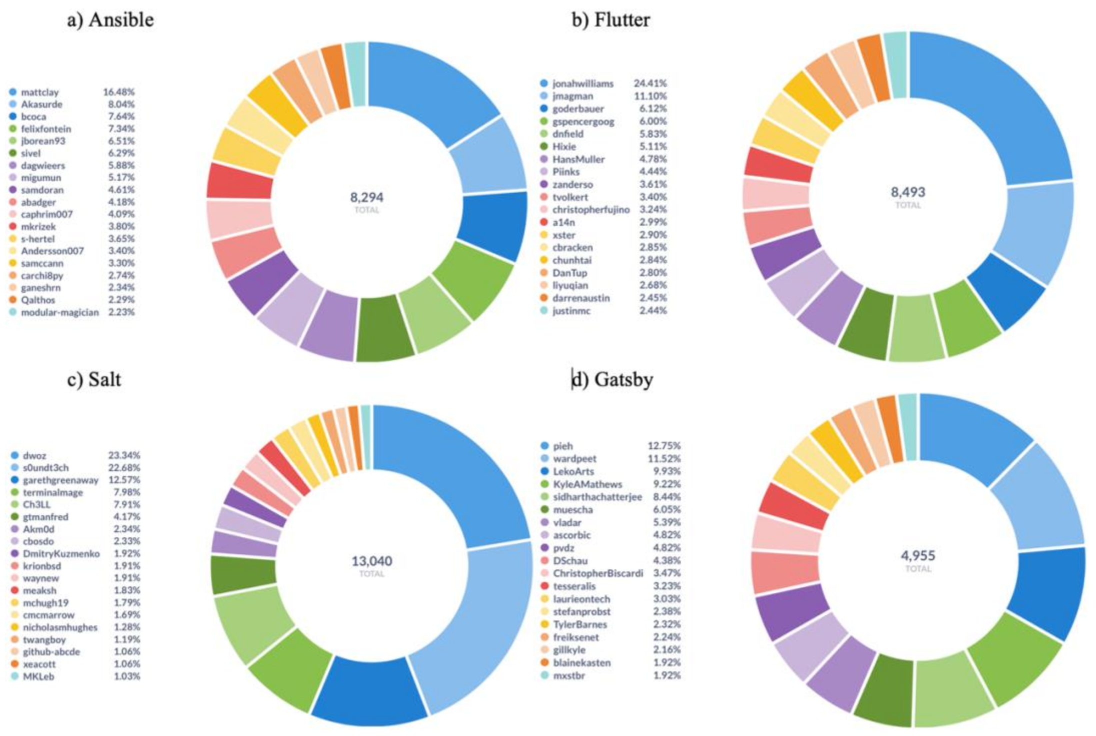


Figure 5.7: Developers with Highest Contributions to a) Ansible, b) Flutter, c) Salt and d) Gatsby

Like ASP.NET Core, Gatsby (created and supported by Gatsby JS), Salt (created and supported by VMware), Ansible (created and supported by Red Hat) and Flutter (created and supported by Google) are also prominently developed and maintained by a group of commercial developers. This proves that the governance model is most likely not the cause for the lack of rhythm in the flow of work.

EQ 3: Do OSS communities have a concept of Adopted Work?

Adopted Work [33] is the work that is moved from the Product Backlog to the Sprint/Release (in this case, project board) after Sprint/Release planning has occurred.

Observation: We can observe that the team is investing time into closing issues that have been around for a significant amount of time, particularly after clusters of shorter-term work, which may suggest a pattern of new product development versus "maintenance mode". This can be interpreted as a modified version of Adopted Work [33], as a result of the team meeting their original forecasted deadline earlier than expected.

Drill-Down: We can verify this by identifying which project these cards belong to and what type of work the project accomplishes. We can begin this inspection by first looking at ASP.NET Core. After filtering the Flow Time based on different projects, we identify that these issues belong to the project board named Servicing. This board is used for tracking all servicing fixes planned for in-support products. In other words, this is the maintenance board for issues which are part of the older LTS release. Further examination reveals that these issues were present in the 6.0.x releases. This shows that the team took the time to finish some of the more dated maintenance work from the 6.0.x releases now that the 7.0.x release development is underway. This can be interpreted as a modified version of Adopted work, which is the work that is moved from the Product Backlog to the Sprint (in this case, project board) as a result of the team meeting their original forecast deadline earlier than expected. The Metabase chart for Flow Time after filtering on project board Servicing is given in Figure 5.8.

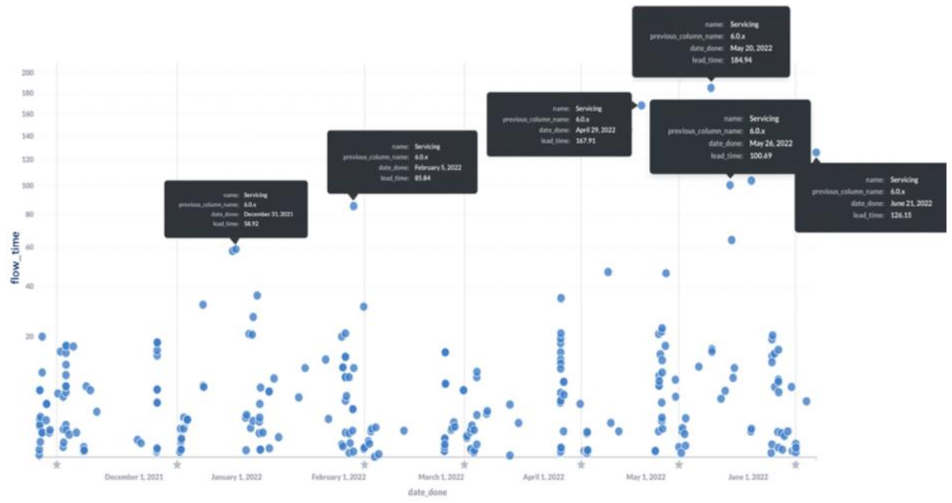


Figure 5.8: Flow Time After Filtering on Project Board Servicing in ASP.NET Core

We consider Gatsby as the next step in the drill down process. Figure 5.9 provides a representation of Aging WIP and Flow Time for Gatsby. It can be easily identified that the reason aging WIP plummeted on January 10, 2022, was because the team closed cards that were around for nearly 3 years. On further investigation, we identify that these cards belong to the project board named Bug Triage. The team uses this project to communicate the order in which they will address the confirmed bugs. A closer look at the communication surrounding the issue reveals that it was resolved in a more recent upgrade to the product, making the earlier issue obsolete. This demonstrates that the team spent the time necessary to clean the backlog. This effort can be considered as a modified version of Adopted Work.

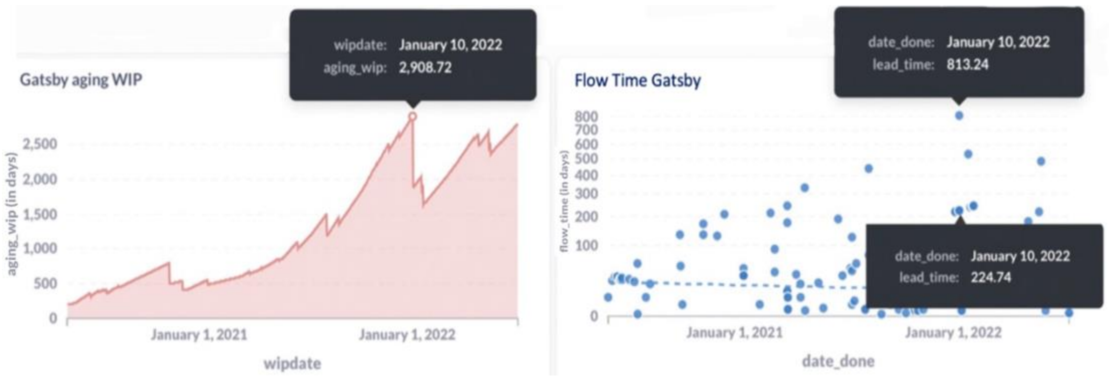


Figure 5.9: Representation of Aging WIP and Flow Time for Gatsby

This investigation aligns with a fundamental difference between OSS and commercial software development. The nature of Adopted Work differs in OSS as such projects are feature-boxed not time-boxed. Given the open-ended “free flow” of value built into an OSS release contrasted with a mature industry-driven team (with a mature cadence and velocity), new work flowing into a release that was not planned at the outset is an expected occurrence, which is validated by this investigation.

EQ 4: Do OSS project boards on GitHub represent the flow of work accurately? Or do they just use the project board as records of past activity?

This EQ presented itself through the visualization of discrepancies between Flow Time and Cycle Time

Observation: There is evidence of a “feature freeze” before OSS releases

Let's now focus on the two most recent releases to determine the flow of work in each release cycle. Figure 5.10 shows releases .NET 6.0.4 and .NET 6.0.5

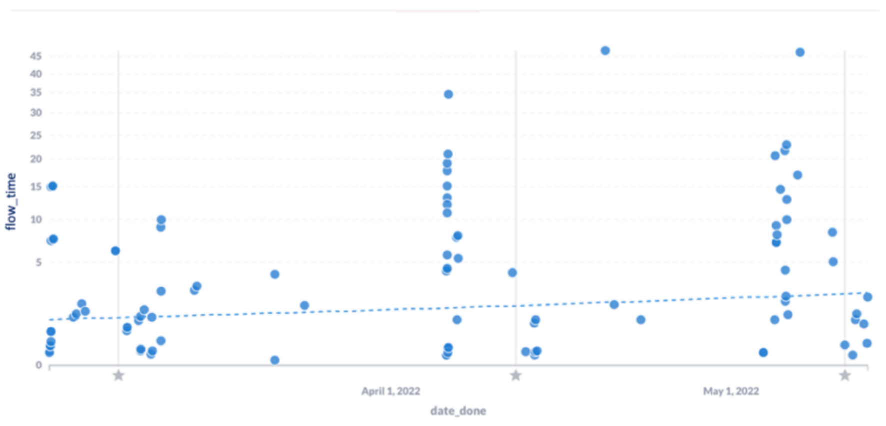


Figure 5.10: Representation of Flow Time for Releases .NET 6.0.4 and .NET 6.0.5

It is important to note that most of the work is completed a little over a week before the release ends. This suggests that before completing the work and releasing it to the client, the team performs some form of "feature freeze". A feature freeze [72] halts all new feature development in order to focus all effort on identifying and fixing bugs and enhancing the user experience. By restricting the introduction of new, untested source code and related interactions, a feature freeze increases the program's stability. Additionally, each release begins with a cluster of issues that have been completed with shorter Flow times, suggesting that most of the work on them was done after the feature freeze of the previous release. Similarly, in their official documentation [73], Ansible discusses the idea of a feature freeze two months before an official release. But in order to conduct additional research, we need a list of all the issues to compare if the card closed after feature freeze is connected to the feature that has already been developed. Considering that not all issues published on GitHub are included in the current data collection method, the data we presently have is insufficient.

Observation: Based on Figure 5.1, it is apparent that in comparison to Flow time, Cycle time has fewer data points. This is problematic as there are several cards that apparently never went to *In Progress* state. There are two reasonable theories in this situation. One, issues added to the project board are directly moved from To Do state to the Done state without entering the In Progress state. Two, issue is added straight to the In Progress state and later moved to the Done state. This shows that the project board is not being used efficiently or even effectively.

Observation: There are various issues with Cycle Time and Flow Time are almost zero, suggesting the board is not representing reality. This discrepancy in flow of work can be verified based on the engagement on the issue.

Drill-Down: Further investigation is based on querying the database to

a) find the number of cards that were directly moved from To Do state to the Done state. This impacts the flow efficiently. There will be no Cycle Time connected with these cards because they were never in In Progress state.

b) number of cards that were directly added to In Progress state. In this scenario, the Flow Time and Cycle Time will be the same.

c) Number of cards whose Flow Time is less than a day. While some of these issues may fairly indicate the amount of effort required to resolve them, the majority of those that were resolved in a single day may not accurately reflect the flow.

Table 5.6: Card Counts Negatively Affecting Flow Efficiency of OSS Projects

Project	**Ansible**	**ASP.NET Core**	**Flutter**	**Gatsby**	**Salt**
Total number of cards in the project board	1053	527	4,284	143	280
Number of cards that never went to the In Progress state	515	68	969	71	43
Number of cards is added straight to the In Progress state	200	250	92	37	9
Number of cards whose Flow Time is less than a day	282	79	218	9	4

The results of this inquiry demonstrate that the approach using which work was tracked in these projects had a flaw. If the work is not adequately tracked, using a project board is futile. A recommendation would be to better diligence in working the process rather than perhaps using the boards as glorified to-do lists or records of past activity.

Observation: Cycle Time exhibits the same increasing trend as Flow Time, indicating that issues are not only on the project boards for an extended period of time, but also remaining in "In Progress" for a significant amount of time. This is consistent with an increase in Aging WIP as shown in Figure 5.11.

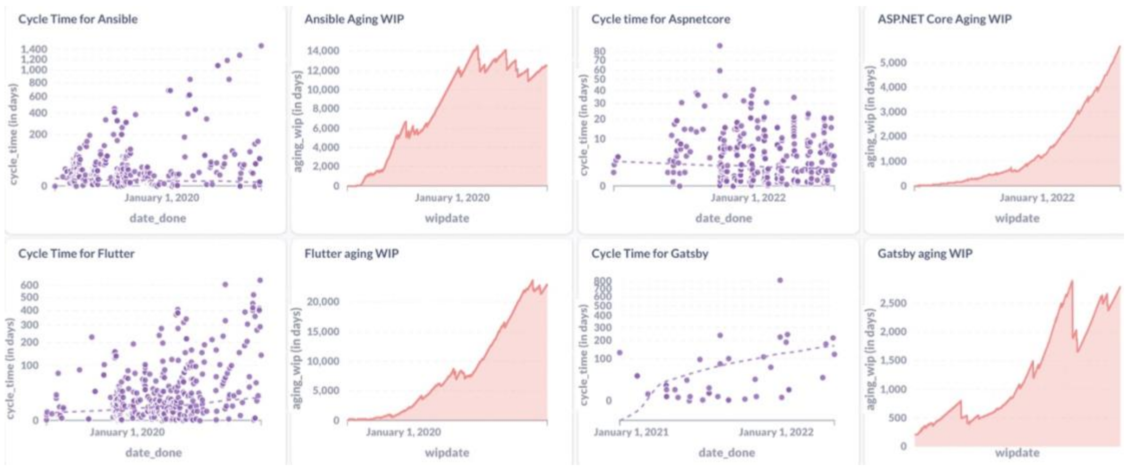


Figure 5.11: Cycle Time and Aging WIP for Ansible, ASP.NET Core, Flutter and Gatsby

The team makes an effort to lower the Aging WIP during each release cycle. The drop in Aging WIP is a result of closed cards. Issues represented in the Cycle time chart indicate that they are in the In Progress state. And when these issues with higher Cycle Time are closed, the Aging WIP illustrates a sharper drop. If we narrow the search to obtain all WIP items with a length of more than six months as of August 2022, each project has the following number of cards a) Ansible - 8, b) ASP.NET – 17, c) Flutter - 36, d) Gatsby - 7, e) Salt – 9.

EQ 5: *Do OSS projects' distribution patterns reveal anything significant about the work being done*?

Examining the flow distributions suggest that as OSS projects age the team takes on incomplete "older" work, or ideas that have been in the issues list for a long time. This reflects on how an Open Source "Product Backlog" is (or is not) refined.

Observation: Scatter plot representation of Flow Time and Cycle Time of Ansible, ASP.NET Core and Flutter is shown in Figure 5.1. The spread of cards we see visually in Flow Time and Cycle Time suggests that 4 of the 5 projects we examined has a cluster distribution pattern. This suggest that the Flow time has larger values as time progress.

Drill Down: These charts in Figure 5.1 contain several issues with lower Flow Times on the left while the Flow Time increases as years progress for Ansible, ASP.NET Core and Flutter. However, the average Flow Time is nearly constant over the years despite the higher Flow Times. The teams in these projects are working more efficiently in 2022, which results in shorter average Flow times than in 2021. In addition to closing numerous recent cards, the teams also invested time closing older cards. This assertion is supported by the descriptive statistics generated for all the included OSS projects as given in Table 5.7. This measure can be used by organizations and volunteers to determine the project's current stage of development and serves as a proof that the project is still active.

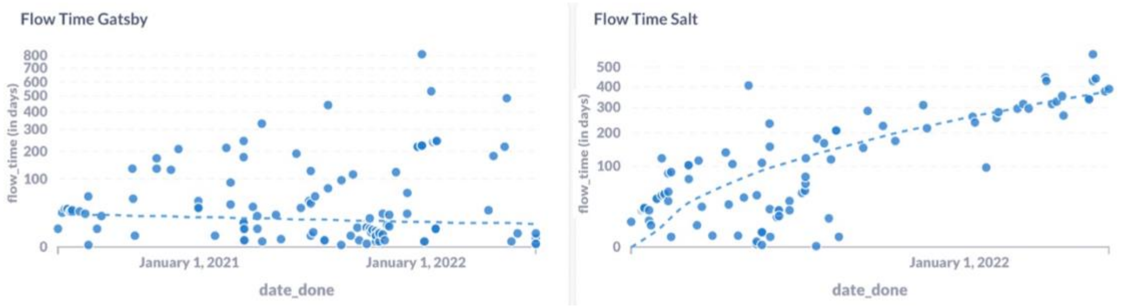


Figure 5.12: Visual Representation of Flow Time for Salt and Gatsby.

Figure 5.12 represents the Flow Time of Salt and Gatsby. The evidence of consistent rhythm in the project mentioned above does not apply to Salt and Gatsby because they operate in a different manner. The average Flow Time in salt grew drastically over time. This could indicate that the project is no longer as active as it once was or that the team is focusing all their efforts on closing older tickets. On the other hand, since there is no strong correlation, no conclusions are drawn about Gatsby based on these visualizations.

To further this analysis, we computed descriptive statistics for each project at each major release point in time. Cycle Time, Lead Time, and Time in Product Backlog were computed, Table 5.6 shows the results for Cycle Time and Lead Time as these are under question here. Visual representation of these descriptive statistics for each of the included project is provided in Appendix D.

Table 5.7: Descriptive Statistics for the Cycle Time and Lead Time for OSS Projects Included

		Cycle Time				**Lead Time**			
Release start date	Release end date	Average	S.D.	Median	IQR	Average	S.D.	Median	IQR
Ansible									
1/12/16	3/23/18	25.94	31.6	13.5	38.5	19.39	33.18	1	21.25
3/24/18	10/4/18	25.06	46.84	9	32	19.53	42.23	4	11.5
10/5/18	5/16/19	70.73	116.6	16.5	61.5	69.46	113.6	17	67
5/17/19	10/31/19	2.31	5.73	0	1	16.78	50.87	1	7
11/1/19	8/13/20	76.7	192.7	1	10.5	259.57	259.6	201.5	531
8/14/20	4/26/21	267.55	335.3	55.5	460.5	434.21	381.6	439	760
4/27/21	11/8/21	428.91	544.2	78	943	373.08	514.3	39.5	901
11/9/21	5/16/22	91.54	296.1	20.5	40.5	261.38	415.5	47	309.5
Gatsby									
11/12/20	3/2/21	51.5	56.39	35	32.5	89.88	85.18	42.5	118
3/3/21	10/25/21	27.41	57.57	3	12.75	48.14	88.8	8	41.5
10/26/21	7/5/22	149.6	204.2	108	193	123	174.5	23	225
ASP.NET Core									
11/11/20	11/8/21	9.05	14.86	4	7.5	18.1	24.15	8	23
11/9/21	6/15/22	7.27	17.83	2	8	17.01	36.54	4	14
Salt									
2/11/20	6/17/20					75		75	
6/18/20	9/2/21	31.13	37.37	20.5	35.25	60.86	74.18	32	83.5
10/23/20	6/21/22	85.92	137.2	28.5	50.25	137.39	143.3	81	221.5
4/1/21	6/21/22	113.41	161.4	28.5	160	169.57	152.3	141	274
Flutter									
2/9/18	11/30/18	14.07	25.51	2	10	32.03	35.92	24	40
12/1/18	11/9/19	33.8	53.07	11.5	27	47.86	83.21	19	35
11/10/19	9/29/20	24.71	42.61	7	26.75	47.47	76.18	16.5	64
9/30/20	3/3/21	63.05	100.1	18	75	103.86	121.3	83.5	129.5
3/4/21	5/9/22	90.87	139.6	27	102.5	160.8	220.1	59	236

This data shows, for all projects except ASP.NET Core, that Cycle Time and Lead Time distribution spread increases, suggesting that a higher percentage of older work is now taken on by the team since the product matured. ASP.NET Core already demonstrated (EQ1 and EQ2) a more consistent approach to flow, and the consistency shown here (albeit only for 2 consecutive major releases) also shows this consistency.

<u>EQ 6:</u> Can visualizations such as Cumulative Flow Diagrams help OSS project teams to find bottlenecks in their process?

The Cumulative Flow Diagram (CFD) [74], provides a concise visualization of the three most important metrics of flow of work: a) Cycle time, b) Throughput, and c) Work in progress. All three metrics are discussed in previous subsections.

Observation: Ideally, CFDs should maintain a low *to-do* and *in-progress* counts while substantially rising *done* counts over the course of the project. This indicates that, in comparison to the amount of work entering the system, more work is being completed. Figure 5.13 provides the CFD for all the OSS projects included in this study.

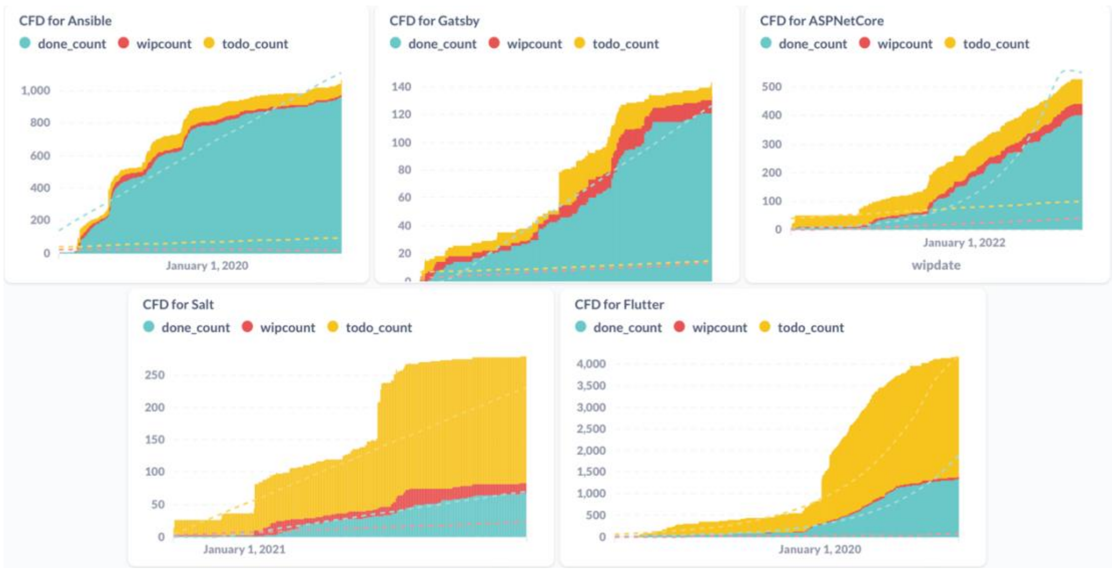


Figure 5.13: CFDs for All the OSS Projects Included in this Study.

Drill Down: Ansible, Gatsby, and ASP.NET Core all exhibit healthy flow efficiency, but Salt and Flutter's flow of work is inconsistent, as can be seen simply by glancing at the CFDs. Without CFDs, this identification would have been considerably more difficult. To comprehend the usefulness of a visualization like CFD, we shall further examine what caused this inconsistency. Starting with flutter, the number of issues added to the project boards are gradually rising, but there is a notable rise on January 11, 2020. We first examine releases to see if there is any data there that can help explain this increase. Releases v1.13.9 and v1.14.0 were made available on January 9 and 14, respectively, 2020. However, there is no mention of the reason for the spike [75]. Next, we look at the issues added to the project boards during January 2020. Based on the results of this investigation, we can infer that a Google developer working on the flutter project added several dated cards to the Awaiting Triage Column in various project boards. Engineers from other disciplines later pick up these cards to review.

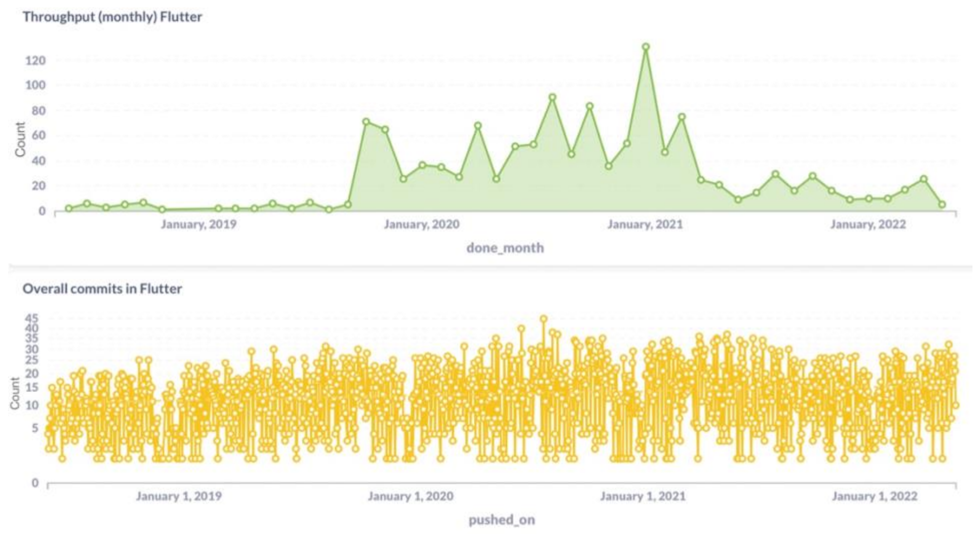


Figure 5.14: Throughput and Overall Commits for Flutter.

There are overall 173 project boards in Flutter. The amount of work added to the project is much higher than the amount of work completed. This indicates that Flutter's throughput is declining over time, as the graph in Figure 5.14 confirms. Yet, we see that there are typically the same number of commits made to the project. This most likely suggests that the dashboard for Flutter does not accurately reflect the project's flow of work. Figure 5.14 displays the throughput and commits for Flutter.

EQ 7: What can be inferred about the success of OSS project's release lifecycle from work completed schedule?

OSS projects sometimes have regular release cycles, but typically they are not time-boxed, and work and value delivery may be sporadic due to external constraints. However, it is useful investigating whether these feature-boxed projects often end up with a fairly regular release cycle, even if unplanned. This would speak to expectations created in the marketplace.

Observation: As Kersten [76] points out, in Open Source projects a huge volume of issues may be proposed that the team simply cannot groom all of those issues effectively. Issues with highest priority is added to the project board after refinement. Product backlog refinement's objective [77] is to keep the backlog filled with work items that are detailed, relevant and in line with the current knowledge about the product and its goals. An OSS project is also subjected to “scope creep” like any other traditional Agile project, in the form of forgotten issues which do not yield substantial value or issues that do not align with project objectives. Without deliberate efforts aimed at managing this inflation, it could lead to the common problems of schedule and budget overruns. Figure 5.15 represents the time each Git issue is created until the time a new card for it is created for that issue.

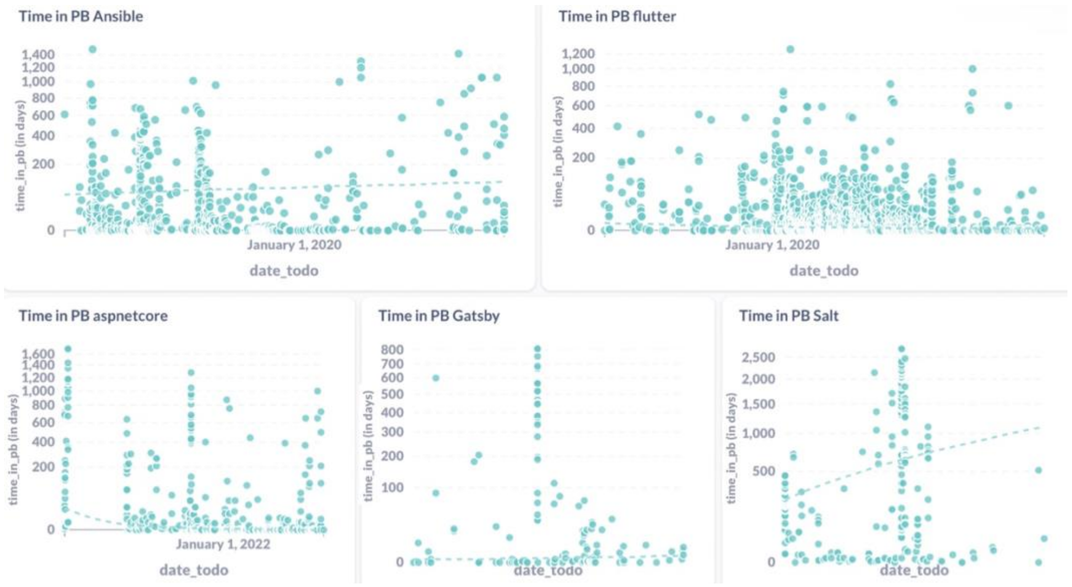


Figure 5.15: Time in Product Backlog for All the Included Projects

These fluctuations in scatter plots are used to represent the addition of new cards to the project board over time. For instance, ASP.NET Core's peak period is represented by the third spike, which occurs in November 2021. A consistent periodic cadence emerged for the project after this point. After this period, the Time in PB is virtually at zero, indicating that the issues that entered the system were almost immediately added to the project board. The last spike denotes Adopted Work, where the team devotes effort to adding issues to the project board that have been present for about a year.

Observation: It appears that several issues were resolved during a small window of time. For example, according to the Flow Time chart of ASP.NET Core (Figure 5.1), since November 2021, several scatter plots have been aligned one over the other approximately every single month, showing that numerous issues were closed almost on the same day. This may suggest that the schedule of a release window may overlap with these collections of cards with short Flow Times, but further research is necessary.

Drill Down: To further investigate the hypothesis of work done within a release window in these projects, releases are plotted on the Flow Time chart. The initial project for this drill down is ASP.NET Core. LTS releases are annotated on the Flow time graph as a first drill down to see if this offers more details. Only one LTS release (.NET 6.0.) coincides with the timeline of the dashboard and unfortunately, this information is insufficient to draw any conclusions. The chart is then highlighted with Current Releases. But both previous Current Release (occurred in November 2020) and the next planned Current Release (November 2022) is outside the range of this research. Adding minor releases (apart from Preview releases) to the Flow time chart is the next step in the drill down process. Figure 5.16 shows all the minor releases (gray vertical lines) marked on Flow Time for ASP.NET Core.

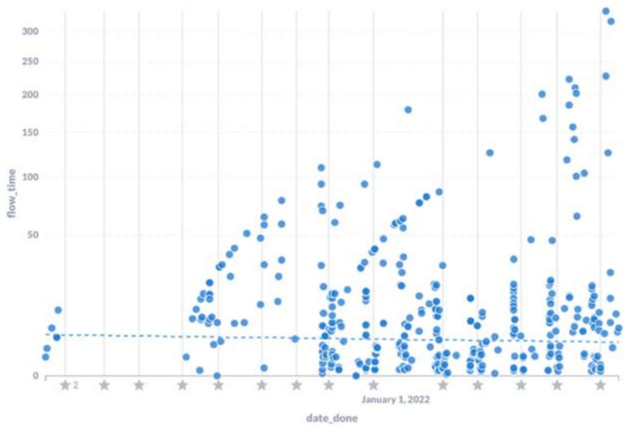


Figure 5.16: Major and Minor Releases Marked on Flow Time for ASP.NET Core.

This filtering offers valuable information. According to this chart, it appears that several issues are completed within a small window for the majority of minor releases. This demonstrates that the project has a successful release schedule planning. The goal of release planning in the Agile approach is to ensure that logical releases happen regularly, and that the product is headed in the right direction [78]. The team is using the project board in the same manner as a typical Agile/Lean project. While some of the issues are completed during the release, most of them are moved to 'Done' near the end of the release. Releases are marked for the rest of the projects to examine if they provide similar information regarding the success of release schedule planning. Figure 5.17 shows all the releases marked on Flow Time.

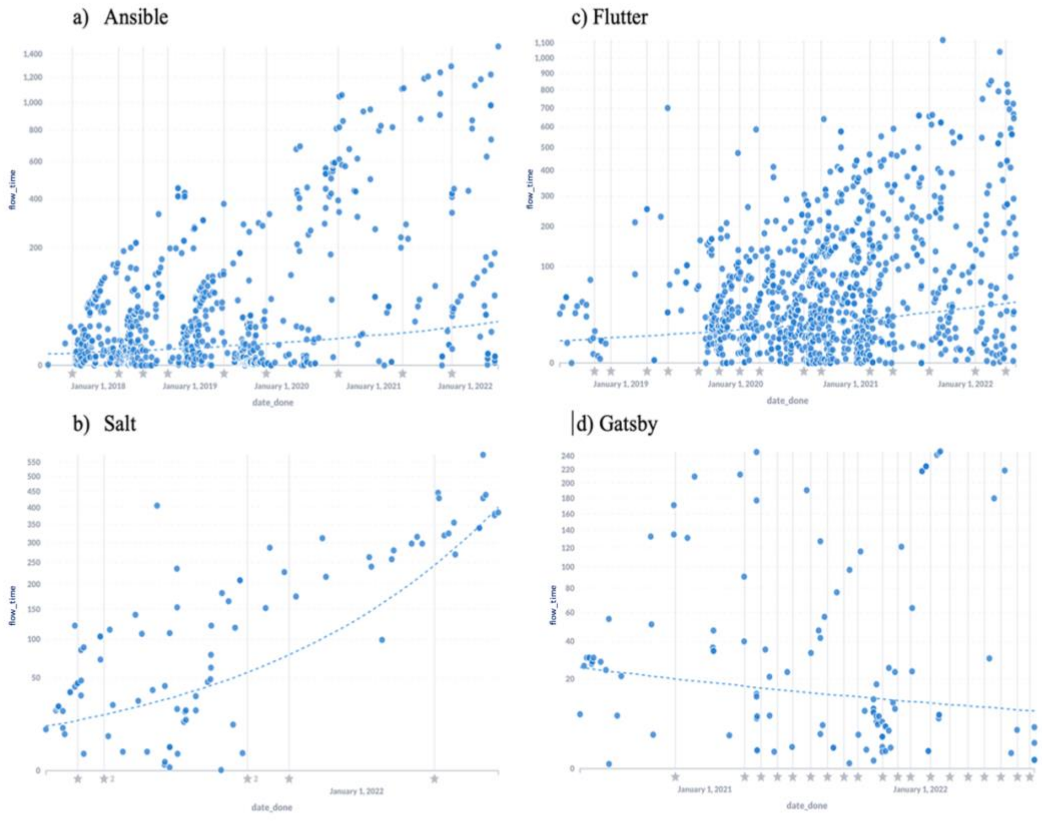


Figure 5.17: Major and Minor Releases Flow Time for a) Ansible, b) Salt, c) Flutter and d) Gatsby

The above graphic shows that Ansible has an effective release planning process as well. We can infer that the release was preceded by some form of ad hoc sprint planning followed by consistent work throughout the release. All planned work was closed within the window of the release. Flutter, Gatsby, and Salt, on the other hand, do not adhere to this pattern. The work distribution within the release appears uncorrelated with the release timeframe. This suggests one of two things: either the team does not consistently adhere to the release dates, or the work performed for the release is not included on the project board.

Observation: The similarity between Lead Time and Flow Time visualizations tells us that the cards were created as soon as the issues entered the system. The Time in PB values for such entries will be close to zero. In such cases both Lead Time and Flow Time will have the same value. But in reality, issues are raised in the user community, this is then triaged and prioritized before entering the project board. This process is known as the backlog refinement. This metric is important as it impacts Aging WIP as Open Source community evaluates when to deliver on issues raised. Figure 5.18 should the comparisons between Lead Time and Flow Time for the included OSS projects.

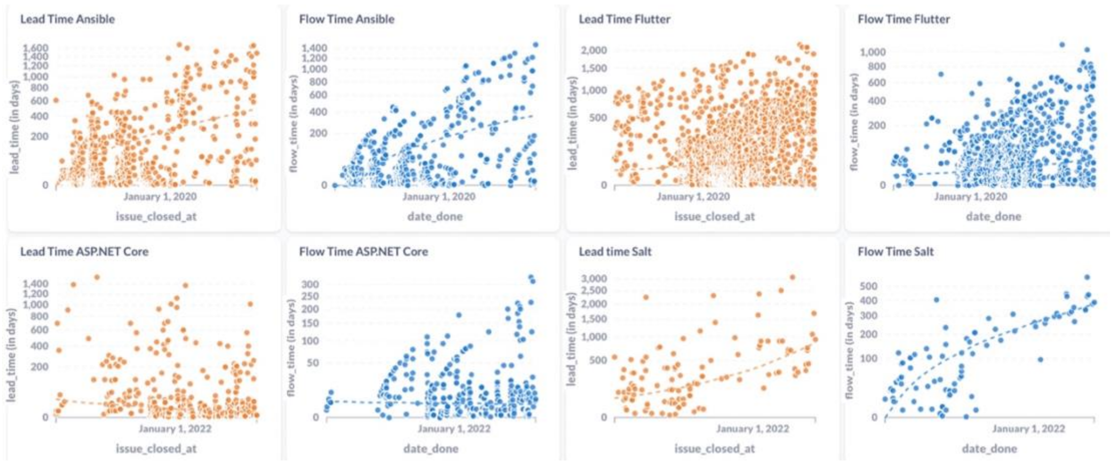


Figure 5.18: Lead Time and Flow Time for the Included OSS Projects.

In terms of the flow efficiency of the included OSS projects, the metrics included in this analysis provide a significant amount of highly relevant information. But it's vital to keep in mind that these investigations and drill downs are not just restricted to the 5 projects covered by this analysis. These projects are merely illustrative. The purpose of this study is to apply these metrics and the knowledge gained to various OSS development projects. The next chapter will go into more detail on how the project's health can be further improved by applying the insights and recommendations gained from these metrics.

5.2 Deductive Reasoning

In addition to the inductive method employed to develop exploratory questions from metric observations, there is also room for us to come up with our own queries about the nature of Open Source and the utility of Agile/Lean metrics in this context. In this section I discuss some investigations I conducted for questions that were of interest.

In Open Source communities, categorization of work is quite intriguing since it reveals how they prioritize the work that provides the most value. In essence, no Open Source projects we have seen attempt to assign value using story points or any other typical Agile measure. I am interested in issue significance and issue type as proxies for value in Open Source projects. I examined Q1 and Q2 presented here in greater detail using a deductive approach to obtain insights into the project.

Q1: How does an OSS community characterize work?

Q2: What is the flow time for high priority work on an OSS project compared to a low priority work? Do Open Source communities exhibit different attitudes towards high priority work?

Addressing Q1: In the OSS projects that included for this study, note that not all the work added in the system is added to the project board. One of the presumptions would be that the work items not included to the project board are less significant than those that are. The idea that the project board's work items are only worked on by core developers may be another factor. Investigating the root of this raises an interesting question about the work prioritization practices used by OSS project teams. While characterizing the nature of work, it is necessary to note that different types of work have distinct values. It is essential to distinguish between work items that are bugs and work items that are new features or product enhancements. The next step is to determine how Open Source communities classify these work items. While there are many distinct levels of value that an OSS project might have, for the purposes of this study it is assumed that all work items are categorized into three levels of severity—high, medium, and low. For instance, major new capabilities of an application differ from incremental improvements in terms of features. Prioritizing work according to their severity is how a product backlog is evaluated.

Determining the severity of bugs and the priority of features is necessary to understand how an OSS project characterizes work. This indicator serves as a proxy for story point estimation since there is no standard way to assign story points in OSS. The Priority/Severity labels defined in Table 5.7 are described as follows:

High Priority/Severity work items are those: a) that requires immediate attention, b) that need to be released immediately after fixed, c) issues/pending fixes that block a release, d) issues that impact most of the customers, e) crashes that terminate the process, f) issue currently blocking a tier-1 customer, and g) bugs that breaks functionality known to work in previous releases.

Medium Priority/Severity work items are those: a) although essential for the release, it could likely be shipped without, b) that is important, but not critical for the release, c) that was better in the past than it is now, d) issues seen by most users, e) issues causing major problems, f) that are incorrect or poor functionality, unclear, and without a workaround, g) issue that impact approximately half of our customers and h) issues that will soon block a tier-1 customer.

Low Priority/Severity work items are those: a) approved but has no time limitation to fix, b) issues considered easy to fix by aspiring contributors, c) issues that impacts only small number of customers, d) fixes or features that are nice to have, e) issues that are likely to work on after high and medium level issues are completed, f) new feature requests, g) that only take a few minutes to fix, and h) cosmetic problems that have a work around.

Table 5.8: Severity and Priority Levels of the Work

	Severity/Priority		
Projects	**High**	**Medium**	**Low**
Ansible	hang, P1	P2	P3, easyfix, small_patch
ASP.NET Core	affected-most, Priority:0, severity-blocking, severity-major	affected-medium, Priority:1, Priority:2, severity-minor	affected-few, good first issue, severity-nice-to-have
Gatsby	No categorization based on priority/severity		
Salt	P1, Regression, severity-critical	P2, severity-high, severity-medium	P3, P4, severity-low, good first issue
Flutter	P0, P1, severe: API break, severe: crash, severe: fatal crash	P2, severe: regression	P3, P4, P5, good first contribution, easy fix

It is clear from the data in Table 5.8 that OSS development communities categorize work by attaching values to it depending on the priority and severity of work. In Q2, it is discussed whether the mindset toward completing such tasks is impacted by their value. Table 5.8 also shows that Gatsby lacks this prioritization of work classification. This absence of information still reveals something about the project and how it operates. It is conceivable that this OSS project failed to exhibit the type of work as they were unaware of it. Considering this, it is advised that OSS project teams become more systematic in their approach to prioritize their work and begin to understand their efficiencies.

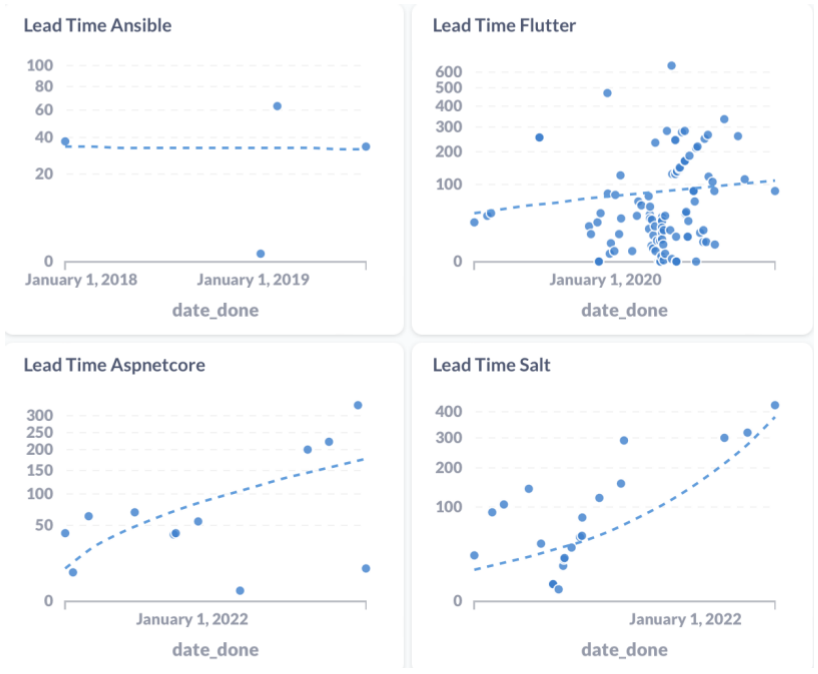


Figure 5.19:Flow Time for High Priority Work Items

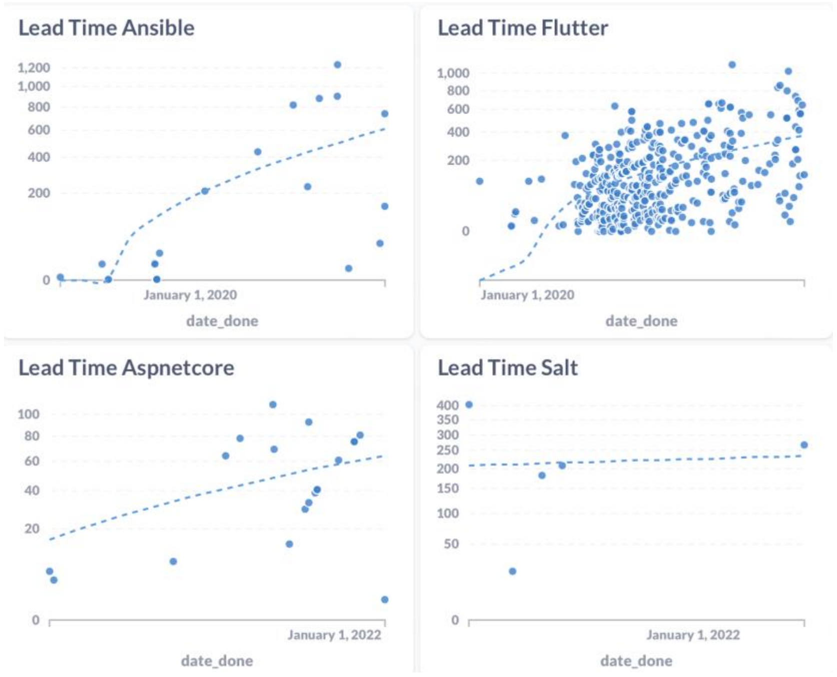


Figure 5.20: Flow Time for Low Priority Work Items

Addressing Q2: The OSS projects present a variety of findings based on the data Figures 5.19 and 5.20 offer. When compared to low priority items, high priority work have much shorter flow times according to Ansible, however for ASP.NET Core, low priority items have shorter flow times. With Salt, there is no obvious distinction in the amount of time work takes based on priority. However, it is crucial to keep in mind that in none of the three cases have sufficient evidence to form firm conclusions. At first appearance, Flutter appears to have lower flow times for high priority work, but the data points for low priority work are noticeably higher, making this comparison unfair. Therefore, while it is true that OSS development communities have a system in place to categorize work according to value, greater care should be taken to ensure that these labels are appropriately applied. An alternative explanation is that the project board may not provide the most accurate picture of all the work that has been assigned a priority. In order to answer this question, we must thus examine all issues; however, as of right now, not all issues listed on GitHub are included in our data collection process. The future scope of work will therefore include retrieving information related to all the issues on GitHub.

To understand the key takeaways from the validation shown in this chapter, the following Discussion chapter emphasizes on the multifarious findings obtained from this work.

CHAPTER 6

DISCUSSION

The previous chapter discusses the results of applying Agile/Lean metrics to the OSS projects included in this study. An inductive approach was used to examine metric outcomes and make observations. To determine what the metric results mean, exploratory questions were created, and drill downs were carried out. Insights drawn from these results are discussed in this chapter, along with the best practices that Open Source communities can adopt to enable better feedback.

The main aim of this study is to examine the usefulness of Agile/Lean software process metrics in OSS development, as well as the limitations associated with their implementation.

6.1 Key Takeaways from This Study:

The results from Chapter 5 demonstrate that Agile/Lean process metrics employed in the context of OSS aid in our understanding of the performance of development teams as well as the effectiveness of flow of work. While the bulk of OSS projects experience values that come in abrupt bursts, it has been observed that some of them show consistent periodicity. We find that the metrics we added can offer helpful feedback if the flow of work and is somewhat consistent.

Revisiting the research questions presented in Chapter 3:

RQ1: Can Agile/Lean metrics provide useful insights about the efficiency of Open Source Software Development communities?

Yes, this study offers numerous insights on the effectiveness of OSS development communities, therefore providing a conclusive positive response to RQ1. The following are some insights produced by this study and reasons why they are important in improving the health of OSS projects. The regularity of the flow-of-work cadence is explained by using metrics like Cycle Time, Flow Time, and Throughput. This is significant because OSS teams can find inefficiencies in their process by comparing these metrics outcomes against the set standards. The uniformity of spikes in the WIP count that occur within each release can provide information regarding the possibility of an informal release planning meeting held during the beginning of a release. Best practices like planning provide clear expectations for all team members by ensuring that their goals are aligned with the project's goals.

For projects with regular throughput cadence, these metrics also aid in determining a release's velocity. By identifying and eliminating bottlenecks, the team will be able to improve velocity with each release, and such release management boosts an organization's number of successful releases while decreasing risk. Flow Type is also used to gain understanding regarding the lifecycle stages adopted by the project. For example, lack of activity in Throughput might suggest the team is taking measures to stabilize the codebase by employing practices such a feature freeze where the code will be thoroughly tested and made bug free before delivering it to the customers. These best practices applied to the projects can serve as a predictor of the quality of the project for the organizations before it is integrated into their system.

Rising trends in Cycle Time and Aging WIP indicates that the project boards may include stale or forgotten work. The fact that OSS initiatives are equally susceptible to scope creep necessitates deliberate measures to manage this inflation. This could lead to the common problems of schedule and budget overruns. These metrics can also be used to get a deeper understanding of the developer community and governance model. This can help volunteers in choosing the right project that is still active and offer support. The descriptive statistics employed in this study allows for data to be presented in a meaningful and understandable way, which, in turn, permits a more straightforward analysis of the study's data set. The teams working on OSS projects can also identify process bottlenecks with the aid of visualization tools like CDFs. Without CFDs, it would have been far more challenging to identify these discrepancies. These are merely a few of the insights presented here. These metrics can be used to a variety of projects to examine how they are interpreted. RQ2 addressed the degree of consistency of various interpretations.

RQ2: Is there a consistent interpretation of Agile/Lean metrics based on the distinct characteristics of Open Source Software projects?

In this investigation, RQ2 was partially resolved. We have a good number of in-depth narratives here that are backed up by data. This entire exercise demonstrates that there is no limitation to the number of exploratory questions we can ask to investigate what additional information this data can reveal. However, going back to the initial research objective, we can see that these measures are significant and that they can be interpreted in a variety of ways. There are several interesting interpretations that may further suggest a potential for a pattern to emerge, albeit we are unsure if there is a consistent interpretation. There are undeniable early encouraging exploratory results, but further study is required to fully comprehend whether we can arrive at a consistent interpretation of these Agile/Lean metrics that can be beneficial across the majority of Open Source projects.

While these metrics provide us with a wealth of useful data, their efficacy can be improved if OSS projects adopt a few strategies. The disparities in the number of data points between Flow Time and Cycle Time indicate that the project board is not being used effectively or even efficiently. Therefore, the first and most recommendation here would be to better diligence in working the process rather than perhaps using the boards as glorified to-do lists or records of past activity. As the work items progress, it is equally crucial to add them in proper states. The metrics result can be substantially hindered by adding work directly to the Done state or skipping the In Progress state. Hence, it is important to move work to relevant states as and when they are completed. The team's use of the project board to track release-related activity is a crucial additional factor. We can observe that a large portion of the work documented in the project boards does not contribute to the release during the release timeframe. This also leads to the expectation that the OSS team either describes their method for refining the Product Backlog or classifies work based on priority and severity using labels, and then assigns these labels with more diligence.

As demonstrated in Chapter 5, Agile/Lean metrics may easily be adapted for OSS projects in many of these situations. For instance, in an OSS context, Adopted Work was adapted to check whether a work item was moved from the Product Backlog to the current project board during a release period. Burndown charts were also altered because the way estimating works in an OSS project differ from how it does in a typical Agile project. This modified burndown chart depicts the progression of work items over the course of a release timeframe. In addition to these, metrics like Fix time for defects [31], Check-ins per day [21], Delivery on Time [18], and more can be modified so that they can adapt for OSS projects, and this can be the focus of future work.

6.2 Limitations of This Study:

There are certain limitations to this study and confounding factors that might affect the authenticity of the findings.

Internal Validity:

Exploratory research is a type of study done on a subject that has not yet been precisely defined. The focus of such research is to grasp and formulate a better understanding of the issue at hand. These research insights cannot be relied upon for effective decision-making as exploratory research brings up tentative results and so is inconclusive.

We just take into account a portion of the Agile/Lean software process metrics in this study. We took the best representative sample of data from GitHub. However, OSS may make use of a variety of different software process metrics, including Velocity, Work Capacity, Found Work, and more. Quality metrics are very important in relation to process flow because defects are a form of waste in Lean [7]. Although we emphasized the value of quality measures, they were not part of the work's intended scope. The overall narrative of issues (reason why most issues were not on the project board, listing all issues that are included in releases, issues core developers worked on, etc.) cannot be comprehended due to limitations in the data collection procedure. Furthermore, some human characteristics, such as the reasons for a higher developer turnover rate, cannot be seen from our datasets.

External Validity:

Although the study included five projects from different organizations, the findings may not be applicable to other OSS project types, such as small scale OSS projects. Besides, a small sample used for exploratory research increases the risk of the sample responses being non-representative of the target audience and may lead to bias while also limiting the applicability of the conclusions

This analysis successfully addressed both research questions. The next chapter provide some pertinent concluding comments while outlining the future score of work in order to summarize the findings presented in Chapter 5 and its related discussion in Chapter 6.

CHAPTER 7

CONCLUSION AND FUTURE WORK

Open Source has significantly altered how businesses create software. Examining the efficiency of Open Source communities is important since OSS projects have a massive economic impact on the market and have recently experienced considerable growth in popularity. Since OSS initiatives frequently adopt similar principles and practices as Agile projects, in this study the usefulness and applicability of Agile/Lean process metrics in OSS is examined, while investigating the opportunities and challenges that may occur when adopting Agile and Lean principles to OSS. The analysis was conducted by applying adapted versions of nine Agile/Lean software process metrics, including Lead time, Flow time, Cycle time, Throughput, Work in Progress, Aging Work in Progress, Cumulative Flow Diagram, Time in Product Backlog, and Adopter Work, to five global OSS projects that have a sizable development team that maintains a mature, well-established codebase with process flow information. An inductive approach was used to examine metric outcomes. This study offers numerous insights on the effectiveness of OSS development communities and also reveals that there are many interesting interpretations that can be inferred. However, at the moment these interpretations proved to be inconsistent.

Given that this research is exploratory, there are numerous potential research opportunities corresponding to this area of study, that are listed here. Impending research on this topic will focus on determining whether it is possible to arrive at a consistent interpretation of these Agile/Lean metrics that can be useful for the majority of Open Source projects. Even in the context of Open Source projects, as quality metrics are crucial, the future scope of work will also include quality measures like escaped defects. Due to restrictions faced in the data collecting process, the overall narrative of issues, including the reasons why the majority of issues were not on the project board, which list of issues were included in releases, which issues core developers worked on, etc., could not be comprehended. In addition to that, understanding the difference between lead time and flow time in terms of metric outcomes will be beneficial. However, to achieve that, a closer look would be required at all the issues listed in the project that were not included in the referenced database. Therefore, retrieving data pertaining to all issues listed on GitHub will be included in the work's future scope. Further, the prospective work may also concentrate on modifying metrics such as Fix time for defects, Check-ins per day, Delivery on Time, Found Work, and more in order to adapt them for OSS projects.

REFERENCES

[1] Atlassian. (n.d.). Learn the essentials of software development. Atlassian. Retrieved October 23, 2022, from https://www.atlassian.com/software-development

[2] Software development statistics - truelist 2022. TrueList. (2022, October 12). Retrieved November 1, 2022, from https://truelist.co/blog/software-development-statistics/

[3] 14th Annual State of Agile Report. Digital.ai. (n.d.). Retrieved November 1, 2022, from https://digital.ai/resource-center/guides/14th-annual-state-of-Agile-report

[4] Manifesto for Agile Software Development. (n.d.). Retrieved November 1, 2022, from https://Agilemanifesto.org/

[5] Stickland, B. (2021, October 26). *Using Agile methodology for software development*. Alliance Software. Retrieved November 1, 2022, from https://www.alliancesoftware.com.au/Agile-methodology-software-development/

[6] Hoory, L. (2022, August 10). *Agile vs. waterfall: Which project management methodology is best for you?* Forbes. Retrieved November 1, 2022, from https://www.forbes.com/advisor/business/Agile-vs-waterfall-methodology/

[7] M. Poppendieck, T. Poppendieck and T. Poppendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley, 2003, ISBN 9780321150783.

[8] Lean thinking and Practice. Lean Enterprise Institute. (2021, August 3). Retrieved November 1, 2022, from https://www.lean.org/lexicon-terms/lean-thinking-and-practice/

[9] *15th Annual State of Agile Report*. Digital.ai. (2022, October 17). Retrieved November 1, 2022, from https://digital.ai/resource-center/analyst-reports/state-of-Agile-report

[10] A comprehensive guide to Agile metrics: Hubstaff tasks. Hubstaff. (n.d.). Retrieved November 1, 2022, from https://hubstaff.com/tasks/Agile-metrics

[11] Wheeler, D. A. (2007). Why Open Source Software/free software (OSS/FS, FLOSS, or FOSS)? Look at the numbers.

[12] Fitzgerald, B. (2006). The transformation of Open Source Software. MIS quarterly, 587-598.

[13] The forrester wave™: Software composition analysis, Q1 2017. Forrester. Retrieved October 31, 2022, from https://www.forrester.com/report/The-Forrester-Wave-Software-Composition-Analysis-Q1-2017/RES136463

[14] *Osposurvey/results_2021.pdf at master · todogroup/osposurvey · github*. (n.d.). Retrieved November 2, 2022, from https://github.com/todogroup/osposurvey/blob/master/2021/results_2021.pdf

[15] Asay, M., Staff, T. R., Branscombe, M., Crouse, M., & Kimachia, K. (2018, October 29). IBM's acquisition of Red Hat Takes Open Source's 2018 tally to nearly $55 billion. TechRepublic. Retrieved October 31, 2022, from https://www.techrepublic.com/article/ibms-acquisition-of-red-hat-takes-open-sources-2018-tally-to-nearly-55-billion/

[16] Koch, S. (2004, June). Agile principles and Open Source Software development: A theoretical and empirical discussion. In International Conference on Extreme Programming and Agile Processes in Software Engineering (pp. 85-93). Springer, Berlin, Heidelberg.

[17] *12 principles behind the Agile Manifesto: Agile Alliance*. Agile Alliance |. (2022, October 11). Retrieved November 1, 2022, from https://www.Agilealliance.org/Agile101/12-principles-behind-the-Agile-manifesto/

[18] K. V. J. Padmini, H. M. N. Dilum Bandara and I. Perera, "Use of software metrics in Agile software development process," 2015 Moratuwa Engineering Research Conference (MERCon), 2015, pp. 312-317, Doi: 10.1109/MERCon.2015.7112365.

[19] S. H. Kan, "Metrics and models in software quality engineering" in, Boston, MA: Longman Publishing, 2002.

[20] Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. IEEE Transactions on software engineering, 28(1), 4-17.

[21] Kupiainen, E., Mäntylä, M. V., & Itkonen, J. (2015). Using metrics in Agile and Lean Software Development–A systematic literature review of industrial studies. Information and software technology, 62, 143-163.

[22] Vesra, A. (2015). A study of various static and dynamic metrics for Open Source Software. International Journal of Computer Applications, 122(10).

[23] Department of Defense, "Agile Metrics Guide, Strategic Considerations and Sample Metrics for Agile Development Solutions", Version 1.1, 23 September 2019.

[24] G. Booch, "Practical Software metrics for Project Management and Process Improvement," Prentice Hall, 1992.

[25] L. Putnam and W. Myers, Ware. "Five Core Metrics: The Intelligence Behind Successful Software Management," Dorset House Publishing, 2003.

[26] K. Pulford, A. Kuntzmann-Combelles, S. Shirlaw, A Quantitative Approach to Software Management: the AMI Handbook, Addison-Wesley Longman Publishing Co., Inc., 1995.

[27] N. Oza and M. Korkala, "Lessons learned in implementing Agile software development metrics", Proc. UK Academy for Information Systems Conf., 2012.

[28] Kersten, M. (2018). Project to product: How to survive and thrive in the age of digital disruption with the flow framework. IT Revolution.

[29] Racheva, Z., Daneva, M., Sikkel, K., & Buglione, L. (2010, June). Business value is not only dollars–results from case study research on Agile software projects. In International Conference on Product Focused Software Process Improvement (pp. 131-145). Springer, Berlin, Heidelberg.

[30] M. Poppendieck, T. Poppendieck and T. Poppendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley, 2003, ISBN 9780321150783.

[31] F. Kišš and B. Rossi, "Agile to Lean Software Development Transformation: A Systematic Literature Review," 2018 Federated Conference on Computer Science and Information Systems (FedCSIS), 2018, pp. 969-973.

[32] Kupiainen, E., Mäntylä, M. V., & Itkonen, J. (2015). Using metrics in Agile and Lean Software Development–A systematic literature review of industrial studies. Information and software technology, 62, 143-163.

[33] S. Downey and J. Sutherland, "Scrum metrics for hyper productive teams: How they fly like fighter aircraft", Proc. 46th Hawaii Intl. Conf. on System Sciences, pp. 4870-4878, 2013.

[34] Kurnia, R., Ferdiana, R., & Wibirama, S. (2018, November). Software metrics classification for Agile scrum process: a literature review. In 2018 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI) (pp. 174-179). IEEE.

[35] Sambinelli, F. and Borges, M. A. F. (2019). The Strategies to Increase Customer Value in Agile: A Survey of Brazilian Software Industry. Journal of Information Systems Engineering & Management, 4(2), em0090.

[36] Anderson, J. C., Rungtusanatham, M., & Schroeder, R. G. (1994). A theory of quality management underlying the Deming management method. Academy of management Review, 19(3), 472-509.

[37] J. Humble, Continuous Delivery vs Continuous Deployment, Mar. 2016

[38] Lead time vs cycle time in Kanban: The Complete Guide. Kanban Software for Agile Project Management. (n.d.). Retrieved October 31, 2022, from https://kanbanize.com/kanban-resources/kanban-software/kanban-lead-cycle-time

[39] The 8 wastes of Lean. The Lean Way. (n.d.). Retrieved November 1, 2022, from https://theleanway.net/The-8-Wastes-of-Lean

[40] Wheeler, D. A. (2006). How to evaluate Open Source Software/free software (OSS/FS) programs (2009).

[41] Feller J, Fitzgerald B: Understanding Open Source Software development Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA; 2002.

[42] *The Open Source definition*. The Open Source Definition | Open Source Initiative. (n.d.). Retrieved November 1, 2022, from http://www.opensource.org/docs/osd

[43] Årdal, C., Alstadsæter, A., & Røttingen, J. A. (2011). Common characteristics of Open Source Software development and applicability for drug discovery: a systematic review. Health Research Policy and Systems, 9(1), 1-16.

[44] Sadowski, B. M., Sadowski-Rasters, G., & Duysters, G. (2008). Transition of governance in a mature open software source community: Evidence from the Debian case. Information Economics and Policy, 20(4), 323-332.

[45] Di Tullio, D., & Staples, D. S. (2013). The governance and control of Open Source Software projects. Journal of Management Information Systems, 30(3), 49-80.

[46] Capra, E., Francalanci, C., & Merlo, F. (2008). An empirical study on the relationship between software design quality, development effort and governance in Open Source projects. IEEE Transactions on Software Engineering, 34(6), 765-782.

[47] Capra E, Francalanci C, Merlo F, Lamastra CR: A survey on firms' participation in Open Source community projects. IFIP International Federation for Information Processing 2009, 225-236.

[48] Hagen S: A comparison of motivation and openness in hybrid Open Source and Open Source Software projects. Master Thesis Norwegian University of Science and Technology, Department of Computer, and Information Science; 2011.

[49] Schweik CM, English R, Haire S: Factors leading to success or abandonment of Open Source commons: An empirical analysis of Source Forge. net projects. The Free and Open Source Software for Geospatial Conference 2008.

[50] David PA, Waterman A, Arora S: FLOSS-US: The free/libre/Open Source Software survey for 2003. Stanford Institute for Economic Policy Research 2004, 20.

[51] Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, Ahmed E. Hassan, and Naoyasu Ubayashi. 2015. Revisiting the applicability of the pareto principle to core development teams in Open Source Software projects. In Proceedings of the 14th International Workshop on Principles of Software Evolution (IWPSE 2015). Association for Computing Machinery, New York, NY, USA, 46–55. https://doi.org/10.1145/2804360.2804366

[52] Lakhani K, Wolf RG: Why hackers do what they do: Understanding motivation and effort in free/Open Source Software projects. In Perspectives on free and Open Source Software. Edited by: Feller J, et al. Cambridge: MIT Press; 2005:3-22.

[53] Tamburri, D.A., Palomba, F., Serebrenik, A. et al. Discovering community patterns in open source: a systematic approach and its evaluation. Empir Software Eng 24, 1369–1417 (2019). https://doi.org/10.1007/s10664-018-9659-9

[54] B. Gezici, N. Özdemir, N. Yılmaz, E. Coşkun, A. Tarhan and O. Chouseinoglou, "Quality and Success in Open Source Software: A Systematic Mapping," 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2019, pp. 363-370, doi: 10.1109/SEAA.2019.00062.

[55] DeLone, W. H., and McLean., E. R "Information Systems Success: The Quest for the Dependent Variable," Information Systems Research (3:1), 1992, pp. 60-95.

[56] Crowston, K., Annabi, H., & Howison, J. Defining Open Source Software project success. In Proceedings of the International Conference on Information Systems (ICIS 2003).

[57] Crowston, K., Annabi, H., & Howison, J. Defining Open Source Software project success. In Proceedings of the International Conference on Information Systems (ICIS 2003).

[58] Ewusi-Mensah, K. "Critical Issues in Abandoned Information Systems Development Projects.," Communication of the ACM (40:9), 1997, pp. 74-80.

[59] Raymond, E. S. "The Cathedral and the Bazaar," First Monday (3:3), 1998.

[60] M. Papamichail, T. Diamantopoulos and A. Symeonidis, "User-Perceived Source Code Quality Estimation Based on Static Analysis Metrics," 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2016, pp. 100-107.

[61] Munaiah, N., Kroh, S., Cabrey, C. et al. Curating GitHub for engineered software projects. Empir Software Eng 22, 3219–3253 (2017).

[62] Openhub.net. Open Hub, the Open Source network. Retrieved October 1, 2022, from https://openhub.net/

[63] B. Kitchenham, what is up with software metrics? – A preliminary mapping study, J. Syst. Softw. 83 (1) (2010) 37–51.

[64] DeGrandis D. (2017). Making Work Visible: Exposing Time Theft to Optimize Work & Flow. IT Revolution.

[65] Raymond, E. S. “The Cathedral and the Bazaar,” First Monday (3:3), 1998.

[66] Zhang, M., Liu, H., Chen, C., Liu, Y., & Bai, S. (2022). Consistent or not? An investigation of using Pull Request Template in GitHub. Information and Software Technology, 144, 106797.

[67] *About projects*. GitHub Docs. (n.d.). Retrieved November 1, 2022, from https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects

[68] M. -G. Jung, S. -A. Youn, J. Bae, and Y. -L. Choi, "A Study on Data Input and Output Performance Comparison of MongoDB and PostgreSQL in the Big Data Environment," 2015 8th International Conference on Database Theory and Application (DTA), 2015, pp. 14-17, doi: 10.1109/DTA.2015.14.

[69] *What is a product backlog?* Scrum.org. (n.d.). Retrieved November 1, 2022, from https://www.scrum.org/resources/what-is-a-product-backlog

[70] Lin, B., Robles, G., & Serebrenik, A. (2017, May). Developer turnover in global, industrial Open Source projects: Insights from applying survival analysis. In *2017 IEEE 12th International Conference on Global Software Engineering (ICGSE)* (pp. 66-75). IEEE

[71] *How to calculate employee turnover rate and what it means*. Built In. (n.d.). Retrieved November 1, 2022, from https://builtin.com/recruiting/turnover-rate

[72] Segal, D. (2022, June 11). *Is code freeze still relevant?* Medium. Retrieved November 1, 2022, from https://medium.com/geekculture/is-code-freeze-still-relevant-9c077495b64

[73] *Core 2.13*. Ansible. (2022, October 18). Retrieved November 1, 2022, from https://docs.ansible.com/ansible/latest/roadmap/ROADMAP_2_13.html

[74] Reinertsen, D. G. (2009). The principles of product development flow: second generation Lean product development (Vol. 62). Redondo Beach: Celeritas.

[75] *Flutter release notes*. Flutter. (n.d.). Retrieved November 1, 2022, from https://docs.flutter.dev/development/tools/sdk/release-notes

[76] Kersten, M. (2018). *Project to product: How to survive and thrive in the age of digital disruption with the flow framework*. IT Revolution.

[77] *What is backlog grooming?* Agile Alliance |. (2022, June 10). Retrieved November 1, 2022, from https://www.Agilealliance.org/glossary/backlog-refinement/

[78] monday.com, A. of us at. (2022, June 15). *Everything you need to know about Agile release planning*. monday.com Blog. Retrieved November 1, 2022, from https://monday.com/blog/rnd/Agile-release-planning/

[79] Leffingwell, D. (2010). *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. Addison-Wesley Professional.

APPENDIX A

LIST OF 40 OPEN SOURCE SOFTWARE COMMUNITIES OBTAINED FROM THE GITHUB REPOSITORY

Sl. No.	**Project Name**	**Domain**	**Presence of Project Board on GitHub**	**Number of Contributors**
1	Amplify-js	Web libraries and fw.	No	345
2	Android	Library	No	34
3	Ansible	Software Tools	Yes	5366
4	Arduino	Rapid prototyping	No	210
5	ASP.NET Core	App. and fw.	Yes	1060
6	Boilerplate	Web libraries and fw.	No	48
7	Bootstrap	Web libraries and fw.	Yes	1300
8	Boto	Web libraries and fw.	No	495
9	Bundler	Web libraries and fw.	No	549
10	Cloud9	Application software	No	64
11	Composer	Software Tools	No	629
12	Cucumber	Software Tools	No	35
13	Cypress	Software Tools	No	344
14	Electron	App. and fw.	Yes	1130
15	Ember-JS	Web libraries and fw.	No	407
16	Flutter	Web libraries and fw.	Yes	1073
17	freeCodeCamp	App. and fw.	No	4424
18	Gatsby	App. and fw.	Yes	3901
19	Gollum	App. fw.	No	143
20	Hammer	Web libraries and fw.	No	84
21	Hawkthorne	Software Tools	No	62
22	Heroku	Software Tools	No	40
23	Kubernetes	App. and fw.	Yes	3194
24	Modernizr	Web libraries and fw.	No	220
25	Mongoid	App. fw.	No	317
26	Monodroid	App. fw.	No	61
27	Netty	Software Tools	Yes	576
28	ohmyzsh	Web libraries and fw.	No	2045
29	PDF-JS	Web libraries and fw.	Yes	361
30	React-Native	Web libraries and fw.	Yes	2319
31	Refinery	Software Tools	No	385

32	Salt	Software Tools	Yes	2366
33	Scikit-Learn	App. and fw.	Yes	2370
34	Scrapy	App. fw.	No	242
35	SimpleCV	App. and fw.	No	69
36	SocketRocket	App. fw.	No	67
37	Storybook	Software Tools	No	1484
38	Swift	App. fw.	No	942
39	TensorFlow	Web libraries and fw.	Yes	3195
40	VS Code	App. fw.	No	1629

APPENDIX B

INCLUSION CRITERIA FOR SELECTED OPEN SOURCE PROJECTS

1. Ansible:

Ansible is a radically simple deployment, configuration management, and ad-hoc task execution tool. It uses the GNU General Public License v3.0 or later [93]. Users can start using it right away, and it supports a wide range of distributions. It also does not require any software to be installed on managed devices and any language may be used to write extension modules. The first lines of source code were added to Ansible in 2012. Recent activity on projects with a code base older than ten years indicates that these projects are likely to tackle critical issues and provide consistent value. Ansible is organized to reward sustained effort by an engaged team of contributors. A long source control history, combined with recent engagement, may imply that this code and ecosystem are significant enough to entice long-term commitment, as well as a mature and reasonably bug-free code base. As of October 1, 2022, 48 developers contributed new code to Ansible in the last one month proving that they have a large, active development team. Ansible is mostly written in Python and 32% of all source code lines are comments, highlighting the project's well-commented source code. A high number of comments might indicate that the code is well-documented and organized, that could be a sign of a helpful and disciplined development team. Over the last twelve months, Ansible has seen a substantial decrease in development activity. This may be a warning sign that interest in this project is waning, or it may indicate a maturing code base that requires fewer fixes and changes.

2. Gatsby:

Gatsby is a static site generator for React. It uses the MIT License [95]. The first lines of source code were added to gatsbyjs/gatsby in 2015. Recent activity on projects with a code base older than eight years indicates that these projects are likely to tackle critical issues and provide consistent value. Gatsby is organized to reward sustained effort by an engaged team of contributors. A long source control history, combined with recent engagement, may imply that this code and ecosystem are significant enough to entice long-term commitment, as well as a mature and reasonably bug-free code base. As of October 1, 2022, 37 developers contributed new code to Gatsby in the last one month highlighting that they have a large, active development team. Gatsby is mostly written in JavaScript and 11% of all source code lines are comments, proving the project has few source code comments indicating that the code is not well-documented and organized. Over the last twelve months, Gatsby has seen a substantial decrease in development activity. This may be a warning sign that interest in this project is waning, or it may indicate a maturing code base that requires fewer fixes and changes.

3. ASP Netcore:

ASP.NET Core is a cross-platform .NET framework for building modern cloud-based web applications on Windows, Mac, or Linux. It uses the MIT License. The first lines of source code were added to dotnet/aspnetcore in 2014. Recent activity on projects with a code base older than eight years indicates that these projects are likely to tackle critical issues and provide consistent value. ASP.NET Core is organized to reward sustained effort by an engaged team of contributors. As of October 1, 2022, 42 developers contributed new code to ASP.NET Core in the last one month proving that they have a large, active development team. ASP.NET Core is mostly written in C# and 15% of all source code lines are comments, indicating the project has few source code comments indicating that the code is not well-documented and organized. Over the last twelve months, dotnet/aspnetcore has seen a substantial decrease in development activity.

4. Flutter:

Flutter framework for building high-performance, high-fidelity iOS and Android apps. It uses the Chromium License [97]. The first lines of source code were added to Flutter in 2005. Recent activity on projects with a code base older than ten years indicates that these projects are likely to tackle critical issues and provide consistent value. A long source control history, combined with recent engagement, may imply that this code and ecosystem are significant enough to entice long-term commitment, as well as a mature and reasonably bug-free code base. As of October 1, 2022, 94 developers contributed new code to Flutter in the last one month highlighting that they have a large, active development team. Flutter is mostly written in Dart and 9% of all source code lines are comments, proving the project is not well-commented. Over the last twelve months, Flutter has seen a substantial increase in development activity. This may be a sign that interest in this project is rising, and that the Open Source community has embraced this project.

5. Salt:

Salt is a unified infrastructure management tool. Building on top of it offers a singular approach to managing the cloud, private, public and multi. It uses the Apache License 2.0 [96]. The first lines of source code were added to Salt in 2011. Recent activity on projects with a code base older than ten years indicates that these projects are likely to tackle critical issues and provide consistent value. A long source control history, combined with recent engagement, may imply that this code and ecosystem are significant enough to entice long-term commitment, as well as a mature and reasonably bug-free code base. As of October 1, 2022, 51 developers contributed new code to Salt in the last one month proving that they have a large, active development team. Salt is mostly written in Python and 30% of all source code lines are comments, highlighting the project's well-commented source code. A high number of comments might indicate that the code is well-documented and organized and could be a sign of a helpful and disciplined development team. Over the last twelve months, Salt has seen a substantial decrease in development activity.

APPENDIX C

SPECIFIC DETAILS OF DATA RETRIVAL FROM THE GITHUB APIS USING A CUSTOMIZED SCRAPER WRITTEN IN PYTHON

Data is made available by GitHub via their REST APIs. With this API, the required information is obtained by sending a request to a distant web server. The simplest way to utilize the GitHub REST API from the command line is through the GitHub CLI. To sign into the GitHub CLI, the auth login subcommand is used. The API request is then made using the Api subcommand. An HTTP method and a path are supplied in a request to the REST API. It is also possible to specify request headers, path, query, or body parameters. The operation path is modified by path parameters. The "Get a repository" path, for instance, is /repos/{owner}/{repo}. The required path parameters are shown by the curly brackets {}. The repository owner and name are given in this instance. The data that is returned for a request can be managed using query parameters. The number of items that are returned when the response is paginated, for instance, could be specified by a query parameter. Additional data can be sent to the API via body parameters. For instance, the "Create an organization repository" function needs a title for the new repository. Additional information is also described, such as text that should go in the repository body. The response status code, response headers, and a response body are all returned by the API. Every request will have an HTTP status code that denotes if the response is successful. A response body is the result of many operations. The response body, unless otherwise stated, is in JSON format. Usually, the REST API provides more details than are required. To extract necessary information, the response can be parsed, if needed.

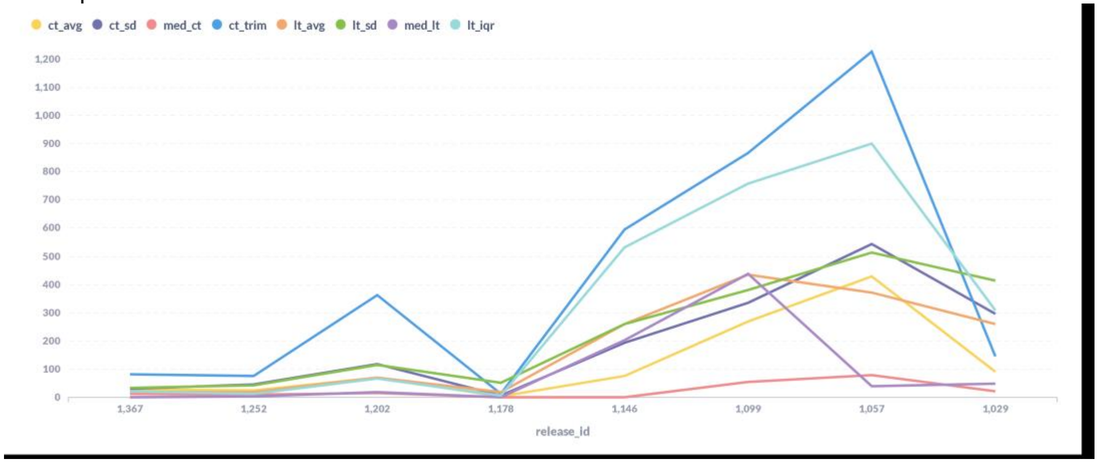
The requests library is the most widely used library in Python for sending requests and interacting with APIs. For the Python programming language, there is an HTTP library called Requests. This is used in the project to provide easier, more approachable HTTP requests. Since the requests library is not a component of the default Python library, it must be installed first. Requests come in a wide variety of forms. Data retrieval is done through the most used one, the GET request. The emphasis will primarily be on making "get" requests because the main task will include retrieval of data. To make a ‘GET’ request, the requests.get() function is used. This requires one argument, that is the URL where the request is made to.

The following GitHub APIs are accessed for each of the included projects to gather information. The python script started by fetching the repository and all the projects (classic) in them. The list projects API returns a 404 Not Found status if projects are disabled in the repository. Insufficient privileges to perform this action, a 401 Unauthorized or 410 Gone status is returned. After that, columns in each of the projects are obtained. Each column is further examined to fetch all the cards in them. Later, the timeline for each of the cards on the project board is fetched. The Timeline events API can return several types of events triggered by timeline activity in issues and pull requests. Every pull request is an issue, but not every issue is a pull request. For this reason, "shared" actions for both features, like managing assignees, labels, and milestones, are provided within Issues API. Along with this, information about the releases for each of the projects is also saved.

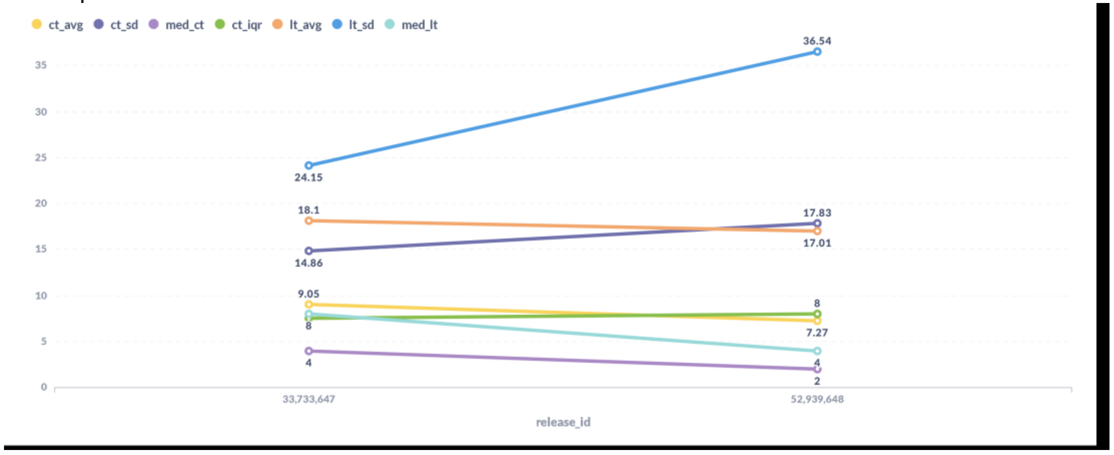
APPENDIX D

LINE CHARTS FOR DESCRIPTIVE STATISTICS OF ANSIBLE, SALT, ASP.NET CORE, GATSBY, AND FLUTTER

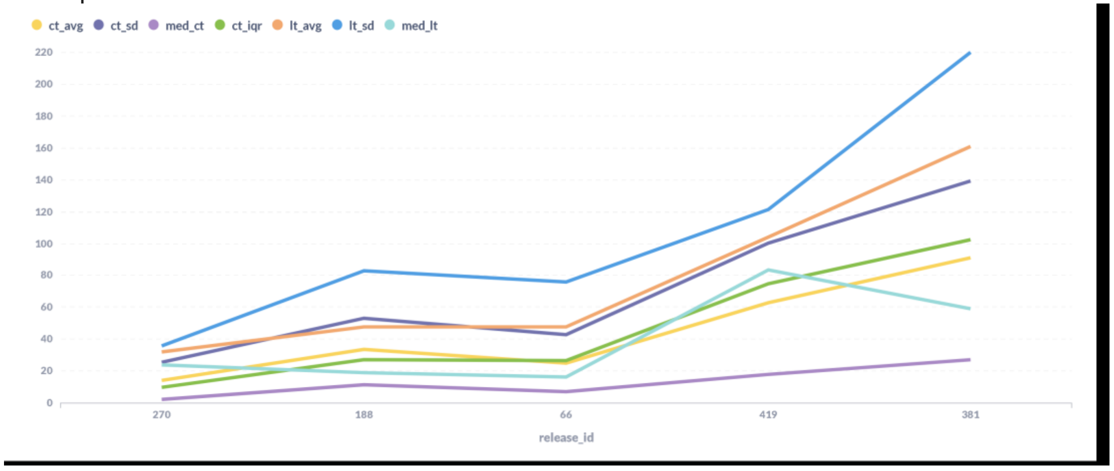
Descriptive Statistics for Ansible



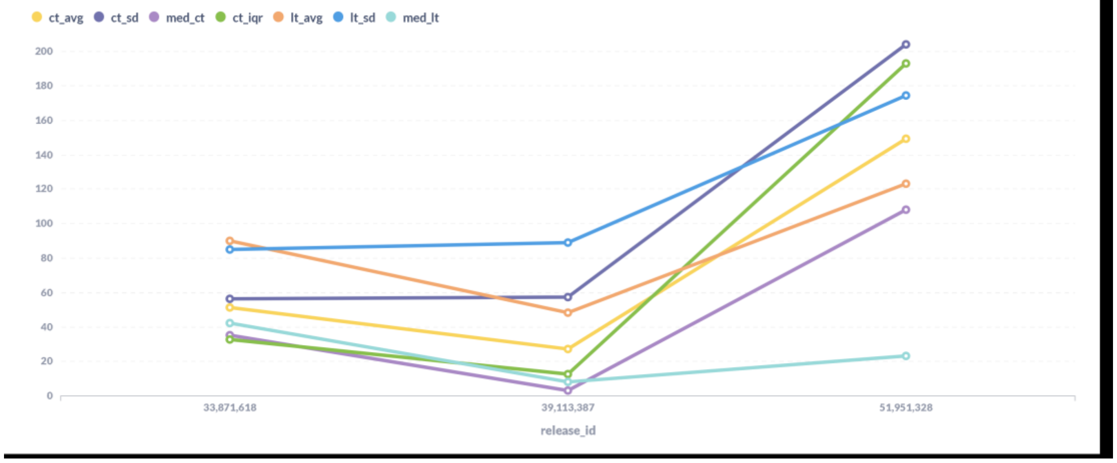
Descriptive Statistics for ASP.NET Core



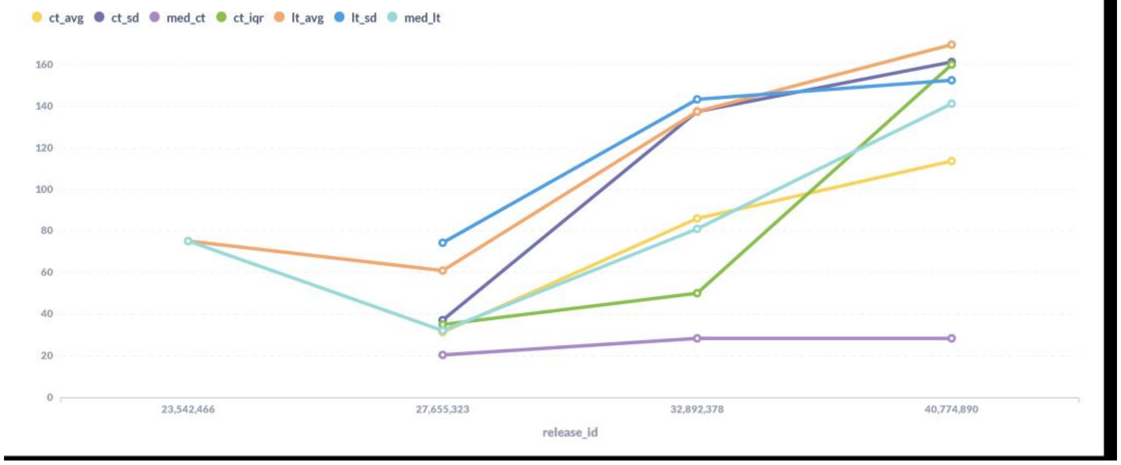
Descriptive Statistics for Flutter



Descriptive Statistics for Gatsby



Descriptive Statistics for Salt


● ct_avg ● ct_sd ● med_ct ● ct_iqr ● lt_avg ● lt_sd ● med_lt
release_id