A Framework for Identifying Evolution Patterns of Open-Source Software Projects

Master’s Thesis

Master of Science in Computer Science

Mattia Bonfanti

Delft University of Technology

Delft, The Netherlands

June 2024

Image Credits: ©2024 Willam Morris. All rights reserved.

A Framework for Identifying Evolution Patterns of Open-Source Software Projects

Master's Thesis

Mattia Bonfanti

A Framework for Identifying Evolution Patterns of Open-Source Software Projects

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Mattia Bonfanti
born in Vaprio d'Adda, Italy

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

©2024 Mattia Bonfanti. All rights reserved.

A Framework for Identifying Evolution Patterns of Open-Source Software Projects

Author: Mattia Bonfanti
Student id: 5002273

Abstract

Research on open-source software evolution gained popularity in the last decade focusing on the theoretical determining factors. Additional works studied growth patterns modeling using time series techniques on small projects and metrics samples or non-openly available larger datasets. Limitations in reproducibility and scalability of these methodologies add to the lack of research on time series methodologies applied to open-source software evolution. Thus, time series approaches from different domains are needed to address the multivariate nature of larger and variable samples of open-source projects and metrics time series data. This thesis aims to provide a reproducible and scalable framework to support researchers in studying open-source software evolution using patterns modeling, time series merging, multivariate time series clustering and multivariate time series forecasting. An openly available dataset of 1328 projects is built using relevant metrics extracted from a systematic literature review. The metrics time series are segmented and clustered to obtain generalized growth patterns: *Steep; Shallow; Plateau*. The sequence of patterns and their correlation are used to create three project clusters, from which prediction models for all metrics are trained to perform multivariate time series forecasting. Experiment results give confidence over the reproducibility and the scalability of the framework and show how the pattern shifts can be linked to real events in projects' histories. The thesis provides an additional perspective on open-source software evolution and can serve as a starting point for further studies.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, Software Engineering, TU Delft
University supervisor:	Dr. Ing. S. Proksch, Faculty EEMCS, Software Engineering, TU Delft
Committee Member:	Dr. JGH Cockx, Faculty EEMCS, Programming Languages, TU Delft

Preface

As this thesis marks the end of my time at TU Delft which started 5 years ago with the Bachelor's in Computer Science and Engineering and continued with the Master's in Computer Science, I cannot stop thinking about the events that led to this. As I was graduating High School in Italy back in 2011, there was great excitement to start University right away at the Polytechnic School of Milan. However, that desire fell short of unfair circumstances that required me to start working only after one year of studies. The desire to get back and pursue a degree at a University was still present, but it could not be a priority anymore. Luckily, I was able to join a small Software Development company in Italy and started learning more about the principles of Software Engineering and how to solve problems through coding. As projects were getting more interesting and challenging in the following 2 years, we had the chance to relocate the business to the USA, specifically in Redwood City (Silicon Valley). The 7 years that I spent there were incredibly fulfilling and helped me to grow as a Software Engineer and as an adult. After achieving things that I am very proud of, the feeling of finishing the University journey that needed to stop years back grew stronger. Thus, I decided to move to TU Delft to make this dream come true. The past 5 years have been challenging between courses, exams and part-time job, however, I am very happy that all the hard work paid off.

A lot of people I met along the way deserve my thanks for making these years so meaningful, but I would like to first give my greatest gratitude to my then-partner and now-wife Emma. She has been incredibly supportive, always stood by my side and reminded me of what was important when doubts started clouding my mind. Additionally, I also want to thank my family and friends for always believing in me and having the right words at the right times.

Many thanks go to my supervisors Sebastian and Shujun, who provided helpful feedback and support during the past months. The members of the thesis committee, Arie and Jesper, are also thanked together with the Delft University of Technology for making this thesis project possible. Special thanks also go to Maria and her family from Maria's Homemade for always providing the most delicious Greek food that fills your heart and soul.

If I had to summarize everything that happened in the past 10-plus years in one sentence, I would choose the following one from the author Daniel Pennac:

In other words, don't panic, nothing goes as planned, that's the only thing the future teaches us when it becomes the past

Dedicated to the ones who are here, the ones who will be here, the ones who were here and the ones who should still be here.

Mattia Bonfanti
Delft, The Netherlands
June 27, 2024

Contents

Preface iii

Contents v

List of Figures vii

List of Tables ix

1 Introduction 1

2 Related Work 5
- 2.1 Open-Source Software Evolution 5
- 2.2 The Phases of Open-Source Software Evolution 14
- 2.3 Time Series Analysis 19

3 Framework Design and Implementation 27
- 3.1 Framework Overview 27
- 3.2 Metrics Selection and Dataset 29
- 3.3 Multivariate Time Series Models 36

4 Results and Evaluation 47
- 4.1 RQ1: What Insights can be Derived from the Framework? 47
- 4.2 RQ2: How Much History is Needed for the Patterns Predictions? 56
- 4.3 RQ3: How Reliable is the Break Points Detection? 60

5 Results Discussion, Limitations and Future Work 83
- 5.1 Framework Insights 83
- 5.2 Patterns Prediction 85
- 5.3 Break Points Detection and Real Events 86
- 5.4 Threats to Validity 87

6 Conclusions 89

6.1 Contributions 89
6.2 Conclusion 90

Bibliography **93**

A Database Schema **105**

B Evolution Patterns Evaluation Plots **109**

C Glossary **113**

List of Figures

3.1 Framework pipeline steps. Source: Thesis author 28
3.2 Time series data of the cumulative stargazers for the saltstack/salt project. Source: Thesis author 37
3.3 Stargazers time series segmentation for the *saltstack/salt* project. Source: Thesis author 39
3.4 Time series segmentation and segments clustering process overview. Source: Guijo-Rubio et al., 2020 [40] 40
3.5 Time series merging procedure. Source: Keogh et al., 1998[57] 42
3.6 Merged metrics time series for the saltstack/salt project. Source: Thesis author 42

4.1 Generalized time series segments growth patterns. Source: Thesis author 48
4.2 Dendrogram representing the hierarchical clustering of open-source software projects based on their metrics evolution patterns. Source: Thesis author 50
4.3 Average metrics evolution patterns for cluster 0. Source: Thesis author 50
4.4 Average metrics evolution patterns for cluster 1. Source: Thesis author 51
4.5 Average metrics evolution patterns for cluster 2. Source: Thesis author 52
4.6 Average patterns prediction performance, deviation and model R^2 scores over training months size evaluated on 10 randomly picked open-source projects. Source: Thesis author 54
4.7 Patterns prediction performance and deviation over increasing N. Source: Thesis author 58
4.8 Patterns prediction performance and deviation over increasing N considering only patterns changes. Source: Thesis author 59
4.9 Merged metrics curve for the patternfly/patternfly-react project. Source: Thesis author 61
4.10 Merged metrics curve for the conan-io/conan project. Source: Thesis author . . 63
4.11 Merged metrics curve for the cockroachdb/cockroach project. Source: Thesis author 65
4.12 Merged metrics curve for the pypa/pip project. Source: Thesis author 66
4.13 Merged metrics curve for the woocommerce/woocommerce project. Source: Thesis author 68

4.14 Merged metrics curve for the nextcloud/server project. Source: Thesis author . 70
4.15 Merged metrics curve for the NixOS/nixpkgs project. Source: Thesis author . . 72
4.16 Merged metrics curve for the WordPress/gutenberg project. Source: Thesis author 74
4.17 Merged metrics curve for the saltstack/salt project. Source: Thesis author . . . 76
4.18 Merged metrics curve for the ansible/awx project. Source: Thesis author 78

A.1 MongoDB collections schema for repositories data and metrics time series data. Source: Thesis author 107

B.1 Generalized time series segments growth patterns for N/2 (664 repositories) input. Source: Thesis author 109
B.2 Generalized time series segments growth patterns for N/4 (332 repositories) input. Source: Thesis author 110
B.3 Generalized time series segments growth patterns for N/8 (166 repositories) input. Source: Thesis author 110
B.4 Generalized time series segments growth patterns for N/16 (83 repositories) input. Source: Thesis author 111
B.5 Generalized time series segments growth patterns for N/32 (41 repositories) input. Source: Thesis author 111

List of Tables

3.1 Keywords and context related synonyms used for the first round of papers search 29
3.2 Questions used to review and summarize a paper with the goal of guiding its quality-assessment score 30
3.3 Quality-assessment questions used to score a paper between 0 and 1. Papers with a score greater than 0.5 were selected 30
3.4 Grouping of open-source software evolution factors cited by surveyed literature 31
3.5 Summary of the open-source software evolution factors with their number of citations in the reviewed literature 32
3.6 Mapping of the open-source software evolution factors to the available metrics from the GitHub API 35
3.7 Evolution factors with the corresponding GitHub API metrics and retrieval endpoints 36

4.1 Generalized patterns curves coefficients for different input size, with full dataset of 1328 repositories as baseline 49
4.2 Distribution of metrics evolution patterns over the repositories clusters 49
4.3 Average metrics patterns prediction accuracy measures and model R^2 score evaluated on 10 randomly picked open-source projects 53
4.4 Average features importance for the metrics forecasting models including how many times they are the most important 55
4.5 Average metrics patterns prediction accuracy measures and model R^2 score evaluated on 10 randomly picked open-source projects using a decreasing amount of features. Features=9 and Months=72 are used as baseline 56
4.6 Bi-grams probabilities and frequencies with EOS indicating one of the ends of a sequence (first or last) 57
4.7 Patterns prediction performance and deviation over increasing N comparison between N-grams and baseline models 57
4.8 Patterns prediction performance and deviation over increasing N comparison between N-grams and baseline models considering pattern changes only 59
4.9 Metrics patterns for the patternfly/patternfly-react project 62
4.10 Metrics patterns for the conan-io/conan project 64

4.11 Metrics patterns for the cockroachdb/cockroach project 66
4.12 Metrics patterns for the pypa/pip project 67
4.13 Metrics patterns for the woocommerce/woocommerce project 69
4.14 Metrics patterns for the nextcloud/server project 71
4.15 Metrics patterns for the NixOS/nixpkgs project 73
4.16 Metrics patterns for the WordPress/gutenberg project 75
4.17 Metrics patterns for the saltstack/salt project 77
4.18 Metrics patterns for the ansible/awx project 79
4.19 Complexity metrics patterns for the patternfly/patternfly-react project 81
4.20 Complexity metrics patterns for the pypa/pip project 81
4.21 Complexity metrics patterns for the cockroachdb/cockroach project 81

Chapter 1

Introduction

Research on open-source software gained popularity over the last decade as the commercial use of open-source components continued to increase [84][73][27][97][102][22][81]. With the emergence of numerous open-source projects, historical resources became available to offer valuable insights into the development of these projects, which made it possible to examine their evolution [34][76][45]. Previous works focused on the aspects that determine the evolution of open-source software by arguing the collection of experiences and building theories of open-source software adoption in terms of planning, process improvement, community involvement and software maintenance [92][86]. These studies relied on qualitative analysis to extract the factors that influence software evolution but lacked the empirical perspective that could give an understanding of the effects of such factors on the evolution of open-source software. Additional studies followed an empirical approach focusing on single large open-source projects and inspected the progress over time of several code-related metrics with the aim of categorizing their growth rate [7][42]. However, the focus of these studies on specific metrics and small amounts of projects raises issues on the scalability of the applied methodologies and derived insights. Although some other studies [61][5] used a larger sample of projects and metrics as input data, the adopted datasets have not been made publicly available, making the results hard to reproduce and expand upon.

Past surveys [56][23] about software evolution studies also pointed out how research on time series analysis generally lacks application to this domain. This prevents time series analysis techniques from being widely applied to the study of software historical data. In previous works on open-source software evolution [61][37][7][5], time series analysis has been used to inspect the independent growth rate of the selected metrics. Additionally, other works focused on time series forecasting by applying the ARMA (Autoregressive moving average model) and ARIMA (Autoregressive integrated moving average) models from the economics domain [38][87][88] to predict future metrics values. Although these works contributed to filling the gap of using time series techniques in the study of open-source software evolution, they limited themselves to analyzing metrics separately and focusing mostly on a single project or a single characteristic (e.g. Java programs). Thus, additional methodologies from other domains where time series research is more prominent must be adopted to obtain reproducible and scalable results in terms of analyzed projects, historical

data and metrics.

This thesis proposes a reproducible and scalable framework of methodologies that supports the analysis of open-source software evolution using time series techniques. The results obtained from exploring the evolutionary behavior of open-source systems serve as a point of reference. This allows the assessment of differences in evolution patterns, the determination of which metrics have an impact on evolutionary behavior and the analysis of the events that might have triggered the growth changes. In addition, modeling the growth rate in open-source projects is used to train models to cluster similar ones and predict future evolution patterns. Given the increasing relevance of research on open-source software evolution, such results would be of high interest to researchers. Additionally, open-source repositories provide a large amount of publicly available data for software engineering studies. The proposed framework is built on more than 1300 project data, which makes it fully replicable and extendable. The historical data of open-source projects is represented by the time series data of the metrics listed below. The metrics are chosen through an extensive literature review of previous work over the past 20 years and the related data is collected using the GitHub API.

- **Commits:** Number of commits over time to the main branch of a project [61][14][7].
- **Releases:** Number of release versions over time [14].
- **Issues:** Number of open issues over time [14].
- **Pull Requests:** Number of open pull requests over time [76][86][13][48].
- **Contributors:** Number of distinct contributors over time [61].
- **Stargazers:** Number of subscribers of a project over time [61].
- **Forks:** Number of created forks over time [35][53][106].
- **Changes Size:** The number of additions and deletions (files, code lines) over time [61].
- **CI/CD Runs:** Adoption of best practices identified by the number of successful CI/CD (Continuous Integration/Deployment) workflows and deployments over time [80][59][74].

The gathered metrics provide the data to perform multivariate time series analysis by applying techniques from different domains (finance and healthcare) to the field of open-source software evolution. In particular, such techniques include growth patterns modeling [40], multivariate time series clustering [9][20] and multivariate time series forecasting [51]. To evaluate the results of the models derived from these methodologies, the following research questions will be answered:

- **RQ1:** *What Insights can be Derived from the Framework?*
  - **RQ1.1:** *What are the Generalized Patterns and Clusters of Open-Source Software Evolution?* This question has the goal of discussing the patterns and clusters generated from the framework methodologies.
  - **RQ1.2:** *How Well Do the Forecasting Models Fit the Historical Data?* This research question aims to assess the quality of the multivariate forecasting models predictions both in terms of future metrics values and future growth patterns derived from the forecast time series.
  - **RQ1.3:** *How do the Metrics Influence Each Other in Project Evolution?* This question aims to provide insights on the correlation of the metrics evolution and how they influence each other.
- **RQ2:** *How Much History is Needed for the Patterns Predictions?* The goal of this research question is to provide insights into how many previous patterns must be known to predict future ones. Predictions of a simple baseline model are compared with the predicitons of a more refined N-grams based model over an increasing amount of known history.
- **RQ3:** *How Reliable is the Break Points Detection?*
  - **RQ3.1:** *How Well Do the Detected Break Points Align With Real Events?* This research question has the goal of confirming the validity of the modeling of the metrics data into a series of evolution patterns by ensuring that a shift in the growth trend is also reflected by a real event around the time of the detected break points.
  - **RQ3.2:** *Does the Metrics Selection Need to be Extended?* This research question aims to inspect if the addition of more metrics can result in the detection of more break points that can be linked to real events.

The answering of the research questions provides insights about the evolution patterns that the projects in the dataset follow (*Steep, Shallow, Plateau*), how the repositories are clustered and how well future patterns can be predicted. The experiment results give confidence over the reproducibility and the scalability of the framework methodologies, which are tested over different sizes of input data in terms of the number of projects, the amount of historical data and the number of metrics. Additional inspections also suggest that the shifts in the metrics evolution patterns can be related to real events in a project's history.

Chapter 2 discusses related work on the topic of open-source software evolution and time series analysis. In chapter 3 the methodology is explained in detail by covering the metrics selection, the creation of the dataset, the patterns modeling, the time series merging, the multivariate time series clustering and the multivariate time series forecasting. The results from the experiments are discussed in chapter 4 where the answers to the research questions are provided. Chapter 5 expands on the results by discussing insights, limitations and threats to validity as well as providing recommendations for future work. Finally, an overview of the contributions and a summary of the presented work are given in chapter 6.

Chapter 2

Related Work

This chapter discusses previous research related to the analysis of open-source software evolution. Section 2.1 argues the factors and the related tangible metrics that influence the evolution of open-source software. In section 2.2 the phases of the open-source software evolution are listed and explained. Finally, section 2.3 illustrates previous work about time series modeling, clustering and forecasting.

2.1 Open-Source Software Evolution

The work by Godfrey et al. from 2000 [34] analyzes software evolution through a case study of the open-source Linux operating system kernel by focusing on both its system and major subsystem levels. Among the factors that contribute to the continuous strong growth of this open-source project, the authors point out how the openness of the licensing impacts the ability of contributors and users to examine the system or change it for their purposes. Additionally, as the nature of open-source software is highly collaborative, the amount of contributors working on a project has a great influence on its progress. This is because a collaborative approach can lead to faster development and innovation, as well as increased community engagement and support. Another relevant factor that the authors discuss is the importance of the user base of open-source software. Since the open-source development model allows for greater freedom and flexibility in terms of personal use and modification of the system, a more user-driven approach to software development is recommended to take into account the preferences of the community.

Another work from Godfrey et al. from 2002 [35] continues the exploration of the open-source Linux operating system kernel evolution through a combination of statistical techniques, case studies, and the development of a specialized tool. The analysis involves examining the growth patterns of open-source software systems using statistical models to understand their sustained super-linear growth. Additionally, the research delves into specific subsystems within the Linux kernel to explore their effects beyond open-source software. Furthermore, the development of the Beagle tool aims to aid in the study of open-source software evolution by integrating various metrics to understand how large systems have changed over time. In discussing this multifaceted approach, the paper highlights

several key factors and metrics that contribute to the evolution of open-source software systems. The number of contributors is a key element that influences the sustained growth, reliability and popularity of open-source software. On top of this, the open-source project footprint, which is seen as its parallel development, also has an impact on its evolution as the offspring might uncover new applications and hidden issues within the base source code.

In their work from 2002, Nakakoji et al. [76] provide insights into the evolution patterns of open-source software systems and communities. The authors propose a classification system for different types of open-source software projects based on their collaboration models and evolution patterns. The paper takes a broader perspective compared to previous studies since, together with the evolution of the open-source software itself, it performs a study on the evolution of the community behind a project as well. Through the case study of several open-source projects, the authors have found that while collaborative development within a community is the essential characteristic of open-source software, different collaboration models exist and have different impacts on the system and its community. From the case studies, the authors highlight different factors and metrics that contribute to the evolution of open-source software projects: number of contributors; number of contributions to the source code (commits); number of users. Overall, these findings provide a better understanding of the role of open-source software communities in driving the evolution of open-source software systems.

The paper from 2003 by Scacchi et al. [86] analyzes open-source software evolution by critically examining Lehman's laws of software evolution [65][66] by evaluating whether they still adapt to contemporary trends and research. As the paper uncovers breakdowns and inconsistencies of the existing laws in the context of open-source software, it advocates the need for potential revisions or alternative ontologies for software evolution to provide a more adequate account that can link theory, practice, and empirical study. The analysis also highlights the need for further study to understand the influence of various variables on the evolution of open-source software systems. Among these factors, the number of involved contributors, their contributions and the number of users are considered the most relevant ones.

In their work from 2004, Paulson et al. [80] analyze software evolution by comparing the evolutionary and static characteristics of open-source and closed-source software systems. The authors quantitatively investigate common perceptions about open-source projects, such as the belief that open-source software grows more quickly, is more modular, fosters more creativity, and has fewer defects. They also examine the hypothesis that external global factors and feedback mechanisms have a greater impact on the evolution of software than the development method. The authors collect and analyze data from three closed-source and three open-source software systems to validate their claims. The main identified factors that influence the growth of open-source software evolution are the size of the community (number of contributors and users) and the development practices.

The paper by Aberdour from 2007 [1] studies the evolution of open-source software from the perspective of software quality. By reviewing existing research on the topic, the author identifies gaps and provides suggestions for further improvements, such as comparing software quality in open-source versus commercial products and studying the impact of

different testing techniques on open-source software quality. The paper highlights several key factors that contribute to the evolution of open-source software systems and communities. The growing number of contributors reflects a sustainable community that fosters rapid code development, effective debugging, and the introduction of new features. On top of this, the adoption of best practices (code quality checks and documentation) facilitates the maintenance and evolution of the software system.

In their work by 2007, Capiluppi et al. [12] expands and refines the empirical hypothesis presented in the staged model of software evolution [83] so that it can be applied to open-source software projects. They analyze each of the phases of the staged model for software evolution and observe when and how differences and commonalities arise in open-source software systems. The main factors that contribute to the evolution of open-source software are identified as the number of releases, number of users, number of contributors and the type of licensing. In particular, new releases tend to be available more often in open-source software, which is seen as a sign of the vitality of the community. This is related to the fact that the users of open-source software can themselves implement fixes that can be delivered in further releases.

The work by Wang et al. from 2007 [101] analyzes software evolution by introducing a new evolution metrics model specifically designed for open-source software and taking into consideration the specific properties of open-source communities. The authors provide a set of metrics for quantitative measuring the evolution of open-source software and conduct a case study on the Ubuntu project using this metrics set. Through this analysis, they aim to provide quantitative evidence of the significance of the open-source communities' evolution. The key factors and metrics that affect open-source software evolution are listed as follows: number of contributors; number of users; number of contributions; number of existing bugs; number of modules in the source code.

A relevant analysis of open-source software evolution is made by Koch in his work from 2007 [61], where he explores the evolutionary behavior of a large sample of open-source software systems. The paper highlights the relationship between the size of a project, the number of participants, and the inequality in the distribution of work within the development team with the presence of super-linear growth patterns. Several key factors that contribute to the evolution of open-source software systems and communities are listed: size of the project; number of users; number of contributions; number of contributions. The paper shows that, while in the mean the growth rate is linear or decreasing over time according to the laws of software evolution, a significant percentage of projects can sustain super-linear growth. These findings can be used to inform the development of open-source software systems and to improve the understanding of the factors that contribute to their evolution.

The work from 2008 by Nakagawa et al. [75] analyzes software evolution by investigating the relationship between software architecture and the quality of open-source software systems. The study focuses on the development of an open-source software web system called Memória Virtual and proposes architecture refactoring activities to improve the maintainability, functionality, and usability of the system. The paper highlights the importance of considering software architecture knowledge and experience in open-source software projects and discusses the occurrence of architecture degradation in open-source software

projects. By analyzing the impact of software architecture on open-source software quality, the paper contributes to the understanding of software evolution in the context of open-source software development. Based on the discussion in the paper, the main factors that contribute to the evolution of open-source software systems and communities are the size and diversity of the community (number of contributors) as well as the adoption of best practices for code quality and architectural choices.

In their work from 2009, Xie et al. [104] investigate software evolution by conducting an empirical study on open-source software. The study focuses on the evolution of seven open-source applications written in C, covering over 69 years of development. The researchers analyze both the development and maintenance branches for each application and find that the growth rate is super-linear on the main development branches and at most linear on the maintenance branches. They also analyze program changes at a fine-grained level and find that the distribution of changes largely follows power laws, with the majority of changes concentrated in a small percentage of code. The study finds that interface changes are much less frequent than implementation changes, and tend to occur towards the initial phases of program evolution. The authors list several key factors that contribute to the evolution of open-source software systems and communities: size of the developer community (number of contributors); adherence to best development practices; ability to respond to users' needs and feedback; ability to evolve and adapt to changing requirements and technologies.

The paper by Hotta et al. from 2010 [46] analyzes software evolution by comparing the modification frequency of duplicate code and non-duplicate code in open-source software systems. The authors use a version control system to obtain the historical data of the source code and identify revisions where one or more source files are modified, added, or deleted. The number of modifications on duplicate code and non-duplicate code are counted to then calculate the modification frequency for both. The authors conduct experiments on 15 open-source software systems and use four duplicate code detection tools to reduce bias. the findings suggest that the presence of duplicate code does not have a seriously negative impact on open-source software evolution. The size of the community, the licensing, the amount of contributions and the adherence to best practices are listed as the main contributing factors to open-source software evolution.

The work from 2010 by Breivold et al.[10] provides a systematic review of published literature on open-source software evolution, with a focus on understanding how software evolvability is addressed during open-source software development. The paper also aims to identify the main research themes and metrics used for measuring open-source software evolution and to discuss the limitations of these metrics. Therefore, the highlighted factors that contribute to the evolution of open-source software systems and communities include the size of the project, the project complexity, the number of contributions and the number of existing issues.

Karus et al., in their work from 2011 [53], analyze software evolution by studying the revision data of 22 open-source software projects over 12 years. The analysis is conducted at two levels: the developer level and the commit level. The developer level investigates the language experience of developers in the projects, examining the commonly used languages and artifact types in open-source software development. On the other hand, the commit level

examines the co-changing files appearing together in commits, identifying co-evolution patterns between different programming languages and artifact types in open-source software projects. It explores the dependencies between file types used in the projects and how these dependencies have changed during the observation period. Among the factors that influence the evolution of open-source software, the number of contributions and the project footprint emerge as the most notable ones.

In their work from 2012, Khan et al. [59] study the evolution of open-source software by reviewing its historical development and impact. The paper discusses the concept of sharing computer programs, the emergence of proprietary software, and the contributions of the GNU Project and the Open-Source Initiative to the evolution of open-source software. Additionally, the paper highlights the growth and widespread usage of open-source software, citing examples such as the number of projects and developers on SourceForge[1] and the download statistics of open-source software applications. This analysis provides insights into the significant transformation and increasing adoption of open-source software over time. The paper identifies several key factors that contribute to the evolution of open-source software systems, such as the number of contributors, the number of users and the adherence to best practices.

Another work by Capiluppi et al. from 2012 [13] performs a comparative study of three different types of open-source software projects to study their evolution: a commercial system (Eclipse), a traditional system (jEdit), and a community project (Moodle). The analysis focuses on the evolution and maintenance activities within these projects to understand the impact of commercial stakeholder involvement. The analysis involves the study of public releases and configuration management systems (CMS) of each system. By examining the type of activities performed by commercial stakeholders and comparing the results achieved by similar open-source software projects with different stakeholder involvement, the paper explores the differences in evolution patterns, maintenance cycles, and complexity management. Additionally, the study utilizes metrics to quantitatively assess the evolution of the studied projects. This quantitative analysis provides insights into the differences in maintenance and evolution activities based on the type of stakeholder involvement, allowing for a comprehensive understanding of software evolution in the context of commercial stakeholder influence. The key factors that contribute to the evolution of open-source software systems and communities include: size and diversity of the stakeholders; project complexity; number of contributions and adoption of best practices.

In their work from 2012, Crowston et al. [23] discuss the evolution of open-source software under two aspects, the software itself and the community that supports the project. Regarding the evolution of the software, the paper cites research that confirms that the evolution of open-source software over time seems to contradict the laws of software evolution [65][66] proposed for commercial software. Regarding the evolution of the community, the paper discusses the dynamic roles of developers and users over time. The contribution made by members is the source of system evolution, and the system evolution, in turn, affects the contribution distribution among the developers. On this note, code and community co-evolve and have an impact on each other. The identified factors that contribute

[1] `https://sourceforge.net/`

to the evolution of open-source software systems and communities include the number of contributions made by members, the number of contributors and the number of users.

The work by Syeed et al. from 2013 [92] inspects the open-source software evolution through a systematic literature review. The paper identifies and categorizes the dimensions of open-source software projects explored under each study facet, such as software evolution, community evolution, and co-evolution. The analysis also includes the examination of research approaches followed in the studies, such as empirical study, case study, comparative study, and tool implementation. This review of previous studies highlights several factors that contribute to the evolution of open-source software: size of the community; adoption of bets practices; project footprint (parallel developments).

The paper from 2015 by Alenezi et al. [2] studies the evolution of open-source software systems complexity. The authors used metrics such as Source Lines of Code (SLOC) and Cyclomatic Complexity (CC) to measure the complexity of the software systems. They then analyzed the growth of ten releases of five well-known open-source projects from different domains to demonstrate how complexity evolves. The study also shows how these systems conform to the second Lehman's law of software evolution, which states that as software evolves, its complexity increases unless proactive measures are taken to reduce or stabilize the complexity [65]. The analyzed factors are the following: lines of code; project size; community size; adoption of best practices.

In their work from 2020, Kumar et al. [63] emphasize the importance of the integration of DevOps practices in open-source software development to enhance security controls and automate processes. The authors argue that the evolution of open-source software is a key component of innovation and continuous improvement of technologies. As open-source software provides source code and community support for innovation and optimization, factors like the number of contributors, number of users, licensing, number of contributions and best practices are highly important in open-source software evolution.

Tandon et al., in their work from 2020 [93], developed mathematical models that predict the number of issues fixed and leftover issues in multi-release open-source software projects to assess their evolution. These models are based on the rates at which different issues are fixed and the rate at which bugs are generated during the fixing of these issues. The analysis demonstrates that the proposed models exhibit promising performance by providing valuable insights into the growth pattern of open-source software and by assisting release managers in making informed decisions based on the number of issues fixed. Based on this study, the main factors that impact open-source software evolution include the size of the community, the level of contributions, the number of releases and the amount of ongoing issues.

The work from 2020 by Baabad et al. [3] presents open-source software evolution from the perspective of software architecture design. In particular, by conducting a systematic study and synthesizing information from previous papers, the authors provide a comprehensive analysis of open-source software evolution and its impact on architectural degradation within the open-source software community. The paper identifies several key factors that contribute to the evolution of open-source software systems and communities. Firstly, the availability of the source code to the public allows for greater collaboration and participa-

tion in the development process. Secondly, the price associated with the value of the system is often lower than that of closed-source software, making it more accessible to a wider range of users. Thirdly, the ability to modify the software to individual needs allows for greater customization and flexibility. Fourthly, the success of open-source software projects has resulted in the stabilization of many researchers and experts that open-source software may extremely contribute to resolving software crises. Finally, the open-source software community has developed a unique methodology for establishing projects that differ from the used method in commercial systems, which has contributed to the success and evolution of open-source software.

In their work from 2020, Dawood et al. [24] addresses software evolution by recognizing the need for a consolidated model for usability evaluation with consistent criteria. The paper emphasizes the importance of integrating all viewpoints into one model to cope with software evolution, indicating that the proposed unified usability evaluation model is designed to adapt to the evolving nature of open-source software. Furthermore, the study's findings and the development of a reliable and validated usability evaluation model contribute to the understanding and assessment of the evolving landscape of open-source software. This approach ensures that the model remains relevant and applicable as software systems continue to evolve. The paper mentions several factors that are relevant to the study's context, including: size of the community; community contributions; best practices adoption.

Bogart et al., in their work from 2021 [8], examine the coordination of breaking changes within open-source software ecosystems. The analysis involves understanding how different ecosystems manage breaking changes, the values that guide their decision-making, and the implications for stakeholders involved in software development. The study employs a mixed-methods approach, combining qualitative and quantitative research methods to analyze software evolution in the context of breaking changes. This includes qualitative coding of survey responses, visualization of survey data, and the identification of patterns and relationships between values and practices across ecosystems. By examining the prevalence of values and practices related to breaking changes across various open-source software ecosystems, the paper provides insights into the evolution of open-source software practices and the aspects that influence decision-making in software development. Furthermore, the paper contributes to the understanding of open-source software evolution by: providing a taxonomy of values and practices related to breaking changes; mapping these values and practices across multiple ecosystems; identifying universal values for software engineers with little variance among ecosystems. In light of the findings, the factors that mostly influence the evolution of open-source software projects are the adoption of best practices, the size of the community and the volume of contributions.

The work from 2021 by Molnar et al. [74] examines the development history of three widely used open-source applications. The study employs several quantitative models to determine patterns in the evolution of open-source software and to understand the rationale behind important changes to the source code. The paper also includes a manual examination of the source code to complement the results from the quantitative models. By analyzing the evolution of open-source software in this way, the paper provides valuable insights into

the maintainability of evolving open-source software and highlights relevant factors that contribute to the evolution itself. The identified factors are related to the size and activity of the community, the number of releases, the adoption of best practices and the amount of ongoing issues.

The research by Michelon et al. from 2022 [72] analyzes open-source software evolution by addressing the challenges of maintaining software variants that evolve in both space and time. The authors introduce a new approach that focuses on locating feature revisions and composing variants with different feature revisions. The analysis involves evaluating the correctness, precision, and recall of the approach in locating feature revisions and composing new configurations. Additionally, the paper computes new metrics for the hints retrieved when composing new products, indicating conflicts and interactions when composing product variants with different combinations of feature revisions. The approach also includes the computation of runtime performance for composing variants with feature revisions, providing insights into the efficiency of the approach in handling software evolution. The main factors that influence the maintainability of an evolving open-source software include the project complexity, the licensing, the size of the community, the amount of contributions and best practices.

Kaur et al., in their work from 2022 [55], study the evolution of open-source software systems and communities over time, identifying patterns and trends in their development. The paper uses a case study approach to examine the evolution of three open-source software projects and their associated communities. The findings of this study provide insights into the factors that influence the evolution of open-source software systems and communities, including the role of community participation and engagement. The key factors that contribute to the evolution of open-source software systems and communities include the size and participation of the community and the best practices adoption.

The work by Zhang et al. from 2022 [106] introduces MyCommunity, a web-based online application system that automatically extracts communication-based community structures from social coding platforms such as GitHub. The system then analyzes and visualizes the community evolution history of an open-source project with semantic-rich events and quantifies the strength of community evolution concerning different events. Additionally, the paper mentions the use of machine learning techniques for predicting project success or failure based on the quantified community evolution events, demonstrating the application's ability to provide valuable insights into the status and future of open-source software projects. The identified factors that influence the evolution of open-source communities include the size of the community itself, the amount of contributions made to the software and the project network.

In their work from 2023, Chakroborti et al. [14] explore the relationship between development and management activities and their impact on the evolution of open-source software. The analysis involves examining the Software Heritage Graph Dataset[2] to extract release information and development records for various open-source software projects. Additionally, the study utilizes GitHub project boards and API[3] to access project manage-

[2]https://docs.softwareheritage.org/devel/swh-dataset/graph/

[3]https://docs.github.com/en/rest?apiVersion=2022-11-28

ment data, including information on issues, pull requests, and customized tasks. By collecting and analyzing data on revisions, releases, and project management activities, the paper aims to understand the patterns of software evolution and the influence of management decisions on the frequency and nature of software releases. Furthermore, the study employs multiple regression analysis to predict the number of monthly releases based on development and management activities, providing a quantitative approach to analyzing software evolution in the context of open-source software development. During the study, the listed key factors that influence the open-source software evolution are the development activities, the best practices adoption, the number of releases, the size of the community and its contributions.

The work by Jain et al. from 2023 [47] illustrates software evolution by employing a historical case narrative of the emergence of the Linux operating system as an alternative to the dominant proprietary software regime. It emphasizes the key role that individuals play in shaping the contours of a technological niche, particularly its identity. The analysis demonstrates how actors amplify a niche through activities such as coalescing and establishing new interaction architectures and practices. Additionally, it specifies how actors are involved in mainstreaming a niche to make it more understandable and acceptable to members of the regime. The sustained coexistence of a niche and regime is highlighted as a distinct form of technology transition, and the impact of the ideology associated with a niche on the larger landscape is explored. By tracing the emergence and development of niche technologies as a social movement, the paper provides a grounded explanation of the processes unfolding as part of technological migration. Community size and involvement are listed as the most important factors that influence open-source software evolution.

Jallow et al., in their work from 2024 [48], analyze software evolution by comparing the change history of code snippets on Stack Overflow[4] with the latest version of code on GitHub. The authors retrieved the set of code snippets that evolved on Stack Overflow since appearing in developer code bases, and that is hence outdated in the GitHub projects. They also applied clone detection to the entire change history of the GitHub code bases and that of Stack Overflow posts to detect whether developers updated their code to a newer version of the Stack Overflow snippet. Finally, for identifying security-relevant edits that have not been transferred to the code bases on GitHub, the authors used a combined method of natural language processing of commit messages and comments and manual confirmation. The study lists the community participation and size, the adoption of best practices and the licensing as the main factors that influence open-source software evolution.

Overall, the extended literature review performed in this section provided significant insights into the factors and metrics that affect open-source software evolution. These metrics serve as a starting point in the building of the open-source projects time series dataset, which is discussed in chapter 3.

[4] `https://stackoverflow.com/`

2.2 The Phases of Open-Source Software Evolution

In their work from 2000, Bennet et al. [6] study software evolution phases in several ways. The authors introduce a new model of software evolution called the ”staged model”, which comprises five distinct steps: initial development, evolution, servicing, phase-out, and retirement. This model provides a framework to analyze software evolution and identify research needs and areas. The paper distinguishes between maintenance and evolution, with maintenance referring to general post-delivery activities and evolution referring to a particular phase in the staged model. The authors argue that software evolution is a distinct activity that requires a different approach than maintenance. The paper also addresses the problems of legacy systems, which are often difficult to maintain and evolve, so a better understanding of such systems is needed to improve software evolution. The proposed staged model of software evolution comprises five distinct phases:

1. **Initial Development**: This phase involves the creation of the first version of the software system, which may be lacking some features but already possesses the architecture that will persist throughout the life of the program. The programming team acquires knowledge of the application domain, user requirements, and other aspects of the software system that will be crucial for subsequent phases of evolution.

2. **Evolution**: This phase involves adapting the software system to changing user requirements and operating environments. The goal is to make substantial changes in the software without damaging the architectural integrity.

3. **Servicing**: This phase involves making small tactical changes to the software system, such as patches, code changes, and wrappers. The software system is no longer a core product, and the cost-benefit of changes is much more marginal.

4. **Phase-out**: This phase involves the gradual reduction of support for the software system, as it becomes less relevant or is replaced by newer systems. The software system is still in use, but its importance is diminishing.

5. **Retirement**: This phase involves the complete retirement of the software system, as it is no longer in use or has been replaced by newer systems. The software system is no longer supported, and its code and documentation may be archived or discarded.

The research by Von Krogh et al. from 2003 [62] proposes several constructs that can help explain how new people join the existing community of software developers in open-source software projects, and how they initially make contributions. These constructs include ”joining script”, ”specialization”, ”contribution barriers”, and ”feature gifts”. The authors suggest that newcomers can derive benefits from specializing in their contributions and that the specialization of newcomers will be related to the contribution barriers in the project. They also propose that feature gifts given by newcomers will be related to their specialization in the project and that these gifts can create new entry points for developers who follow. These constructs offer insights into the processes and strategies that can impact

the evolution of open-source software systems and communities. From this, the following evolution phases are presented:

1. **Initial Development**: This phase involves the initial creation of the software, including the design, coding, and testing of the first version.

2. **Maintenance and Bug Fixing**: After the initial release, the software enters a phase of maintenance where bugs and issues are identified and fixed.

3. **Updates and Enhancements**: As the software is used, new features and enhancements may be added to improve its functionality and address user needs.

4. **Legacy Support**: Over time, the software may become a legacy system, requiring ongoing support and maintenance to ensure its continued operation.

5. **End of Life**: Eventually, the software may reach the end of its useful life and be retired or replaced by newer systems.

The work from 2006 by Girba et al. [43] analyzes software evolution phases by introducing Hismo, a meta-model that adds a time layer on top of structural information. Hismo is designed to provide a common infrastructure for expressing and combining evolution analyses and structural analyses. The paper demonstrates how various software evolution analyses can be expressed using Hismo, including measurements for quantifying changes, reverse engineering analyses, historical co-change, and class hierarchy evolution visualization. Overall, the paper provides a framework for understanding and analyzing the evolution of software systems, enabling better reasoning about software systems and the derivation of general laws of software evolution based on historical data. The identified phases of software evolution are listed as follows:

1. **Initial Development**: This phase involves the creation of the initial version of the software, including requirements gathering, design, implementation, and testing.

2. **Maintenance**: After the initial release, the software enters the maintenance phase, where updates, bug fixes, and minor enhancements are made to address issues and improve functionality.

3. **Evolution**: As the software continues to be maintained, it evolves to meet changing user needs, technological advancements, and market demands. This phase involves significant changes, new feature additions, and architectural modifications.

4. **Retirement/Phase-out**: Eventually, software reaches the end of its life cycle and may be retired or phased out. This phase involves transitioning users to alternative solutions and discontinuing support for the software.

These phases are not always linear and may overlap, especially in the case of long-lived software systems. Additionally, the evolution phase may involve iterative cycles of

maintenance, updates, and enhancements as the software continues to adapt to changing requirements and environments.

Fluri et al., in their work from 2007 [31], focus on the identification and extraction of particular changes that occur across multiple versions of open-source software. The authors introduce a change-distilling algorithm that enables fine-grained source code change extraction by improving upon existing methods. By evaluating the algorithm with a benchmark of manually classified changes in revisions of methods from open-source projects, the paper demonstrates significant improvements in extracting types of source code changes. This analysis contributes to a better understanding of software evolution by providing a more accurate and detailed approach to identifying and categorizing changes in source code over time. The phases of software evolution can vary depending on the specific model or framework used, but generally include the following:

1. **Inception**: This phase involves the initial planning and conceptualization of the software system, including defining the scope, requirements, and goals.

2. **Development**: In this phase, the software system is designed, developed, and tested. This includes coding, testing, and debugging to ensure that the software meets the specified requirements.

3. **Maintenance**: Once the software system is deployed, it enters the maintenance phase. This involves ongoing support, bug fixing, and updates to ensure that the software remains functional and relevant.

4. **Evolution**: The evolution phase involves the ongoing development and improvement of the software system over time. This can include adding new features, adapting to changing requirements, and addressing issues that arise during maintenance.

5. **Retirement**: Eventually, the software system may reach the end of its useful life and be retired. This may involve transitioning to a new system or archiving the software for historical purposes.

The paper by Godfrey et al. from 2008 [36] discusses the historical context of software evolution, and the current state of research in the field, and outlines future challenges and opportunities for software evolution research. The paper aims to highlight the importance of understanding how software systems evolve and the implications of this evolution on software development practices. The work involves a literature review and analysis of existing research in the field of software evolution. The authors discuss insights from Lehman's laws of software evolution [65], the staged life cycle model of Bennett and Rajlich [83], and other relevant concepts to provide a comprehensive understanding of software evolution. According to the findings of the literature review, the lifespan of a typical software system is divided into four stages:

1. **Initial Development**: In this stage, the first version of the software system is developed. Knowledge about the system is fresh and constantly changing, with change

being the norm. An architecture emerges and stabilizes, laying the foundation for future development.

2. **Active Evolution**: During this stage, simple changes can be easily implemented, and more significant changes are also possible, albeit with increased cost and risk compared to the initial development stage. Knowledge about the system remains good, although many original developers may have moved on.

3. **Servicing**: In the servicing stage, the software system is no longer a key focus for developers, who primarily concentrate on maintenance tasks to keep the system running. Architectural or functional changes take a back seat, and the predictability of change decreases as knowledge about the system diminishes.

4. **Phase Out**: This phase is characterized by the decision to replace or eliminate the system, either because the maintenance costs have become too high or because there is a more suitable solution to be deployed. An exit strategy is planned and implemented, often involving techniques such as legacy wrapping and data migration. Ultimately, the system is shut down.

The research by Guimaraes et al. from 2013 [26] investigates the life cycle patterns of open-source software development communities (OSSDC) using functional data analysis. The study aims to understand how OSSDC evolves in terms of activity levels and effectiveness, providing insights into the dynamics of these communities and their development stages. Effectiveness levels are measured by the number of downloads in a month, while activity levels are measured by various actions such as bugs opened, messages posted and code contributed. The data is transformed to make values commensurate and adjusted for project size. Functional data analysis is employed to assess hypotheses regarding the shapes of effectiveness and activity levels over time. This finding emphasizes the importance of understanding the underlying processes of community activity and participation dynamics for enhancing community effectiveness and survival. Such considerations are relevant for online community users and policymakers. Users are encouraged to pay greater attention to community activity and participation dynamics, while policymakers gain insights into the development patterns and the relationship between effectiveness and activity levels in OSSDC. The identified phases of software evolution typically include the following stages:

1. **Initial Development**: This phase involves the initial creation of the software product, including defining requirements, designing the architecture, coding, and testing.

2. **Growth**: During the growth phase, the software product gains users and features. Updates and enhancements are made to meet user needs and address any issues that arise.

3. **Maturity**: In the maturity phase, the software product has stabilized, and the focus shifts to maintaining and optimizing the product. This phase involves regular updates, bug fixes, and support for existing users.

4. **Decline**: The decline phase occurs when the software product becomes outdated or faces competition from newer technologies. Sales and user base may decrease during this phase.

5. **Retirement**: The retirement phase marks the end of the software product's life cycle. The product is no longer supported, and users are encouraged to migrate to newer solutions.

Barahona et al., in their work from 2014 [37], aim to explore the applicability and validity of the laws of software evolution in the context of a real-world, long-lived software project. Additionally, the paper aims to develop a methodology for analyzing the evolution of large, long-lived software projects using data from Software Configuration Management (SCM) systems, providing insights into how such projects evolve. This study provides a detailed analysis of 20 years of the glibc project, offering insights into its evolution patterns over time. It is one of the first studies to examine the life of a software project over such an extended period using data from its SCM repository. This analysis is done by introducing a methodology for studying the evolution of long-lived software projects. This methodology outlines specific steps for data retrieval, validation, and analysis to interpret the laws of software evolution in the context of a project's history. The adopted transparency enhances the reproducibility of the study and provides a framework for future research in analyzing software evolution in large projects. The phases of software evolution, as identified in the study, are based on the staged model of software evolution [6]. This model divides the lifetime of a software project into five stages:

1. **Initial Development**: This phase involves the initial creation and development of the software product until its first release. It encompasses activities such as design, coding, and testing to bring the software to a functional state.

2. **Evolution**: After the initial release, the software enters the evolution phase, where it undergoes continuous updates, enhancements, and modifications to meet changing requirements and address issues discovered during usage.

3. **Servicing**: In the servicing phase, the focus shifts towards maintaining the software by providing patches, updates, and fixes to address bugs, security vulnerabilities, and other issues without introducing new features.

4. **Phase-Out**: The phase-out stage occurs when a new version or system is introduced to replace the existing software. This phase involves transitioning users to the new system while gradually phasing out the old one.

5. **Close-Down**: The close-down phase marks the end of the software's lifecycle, where the system is discontinued, and resources are reallocated to other projects or initiatives. This phase involves archiving data, documenting final processes, and ensuring a smooth transition to alternative solutions.

The phases of software evolution described in this section highlight the changing nature of software systems over time and the different challenges that come into play. The information highlighted in this literature review provides the basis to link the evolution patterns to the theoretical concepts of previous research. This is discussed in chapter 5.2.

2.3 Time Series Analysis

In this section, previous research on the analysis of time series data is presented by discussing the time series modeling in section 2.3.1, the clustering in section 2.3.2 and the forecasting in section 2.3.3.

2.3.1 Time Series Modeling

The work by Han et al. from 1998 [44] discusses new data mining algorithms by augmenting their methodologies, real-world applications and effectiveness through validation experiments. The main focus of these techniques is the processing of time series data to extract patterns, trends and insights that can help in understanding the underlying dynamics of the data. The findings can then be applied to perform clustering, classification and association of the time series data. Overall, the algorithms presented in this work allow for the identification of temporal patterns and relationships that can be valuable for making predictions and informed decisions in various domains.

Guranlik et al, in their work from 1999 [41], present an approach for change point detection in time series data. This involves detecting changes in the model or parameters that describe the underlying data, without assumptions about existing deviation points. The paper combines change point detection and model selection techniques to develop algorithms for both batch and incremental versions of the problem. The goal is to provide a method that is robust against noisy data sets and outperforms visual inspection by humans. The approach starts with the modeling of time series data by representing it using a set of statistical parameters that are used to identify change points in time segments through likelihood criteria. Significant shifts in the underlying model will represent a significant event at a given point in the time series. The effectiveness of the approach is assessed by comparing it with visual inspection in noisy time series data. The benefit of the introduced methodology is the enabling of event detection in environments where the underlying phenomenon is not well understood.

The paper by Keogh et al. from 2001 [58] provides an extensive review and comparison of algorithms for segmenting time series data, with a focus on introducing a new scalable and high-quality approximation algorithm. Through empirical evaluation of diverse datasets, the authors address the limitations of existing approaches and show the effectiveness of the new one. The introduced algorithm called SWAB is a combination of the existing Sliding Window and Bottom-up approaches. In particular, the goal is to include the online nature of Sliding Window to the highly robust segmentation performance of the Bottom-up algorithm.

In their work from 2004, Chung et al. [20] introduce a new distance measure for evolutionary time series based on the similarity of segments' patterns. The paper focuses on improving the performance of evolutionary approaches by addressing the limitations of existing distance measures, specifically the Direct Point-to-Point Distance (DPPD) approach. The proposed new distance compares the trend similarity of two sequences, making it robust to amplitude transformation, time phase, scale, and baseline differences. This will enhance the accuracy of time series segmentation by utilizing pattern distance as a more effective evaluation metric compared to traditional methods. In particular, the segments are identified through the perceptually important points in the time series, which capture essential characteristics of the data. Experimental evaluation supports the claims that the new pattern distance performs better than the existing DPPD.

The work by Sabeti et al. from 2020 [85] introduces a novel method called "pattern tree for learning patterns in time series" data using a binary-structured tree. The paper focuses on the application of the pattern tree method in time series estimation and forecasting, aiming to improve the mean squared error of estimation compared to other methods. Additionally, the paper discusses the potential applications of the pattern tree method in handling continuous streams of information from IoT devices and sensors, highlighting its versatility for tasks such as lossless compression, prediction, and anomaly detection in time series data. The analysis of time series data involves the following key steps: pattern identification; pattern learning; pattern estimation; pattern forecasting. The paper conducts a comparative analysis of the pattern tree method with other existing approaches, such as linear prediction and pattern-based forecasting. Overall, the introduced algorithm successfully improves estimation accuracy and enables real-time analysis of evolving data streams.

Among the described work about time series modeling, the techniques from Keogh et al. [58] and Chung et al. [20] were picked to be included in the presented framework. Keogh et al. [58] provide a reproducible and easy-to-apply algorithm to merge time series into a unique representation. Chung et al. [20] introduce a novel distance measure that focuses on time series patterns and it can be translated from the financial domain to the topic of open-source software evolution.

2.3.2 Time Series Clustering

In their work from 2006, Wang et al. [100] propose a method for clustering time series data based on their structural characteristics rather than distance metrics. The method aims to extract global features from time series data, such as trend, seasonality, periodicity, and other characteristics, to reduce dimensionality, improve robustness to missing or noisy data, and provide meaningful clusters for analysis. Feature extraction and dimensionality reduction make the process more manageable for clustering algorithms, especially when dealing with sizeable time series datasets.

Zhang et al., in their work from 2011 [108], introduce a novel algorithm for shape-based time series clustering that can reduce data size, improve efficiency, and maintain effectiveness by utilizing complex network principles. The algorithm involves building a one-nearest neighbor network based on time series similarity, selecting nodes with high de-

grees for clustering, and applying dynamic time warping distance function and hierarchical clustering. The analysis involves: common properties extraction (noise, amplitude scaling, and temporal drift); time series distance evaluation using dynamic time warping (DTW); building 1-NN network based on series similarity; identifying nodes with high clustering density; performing hierarchical clustering on nodes of neighbors. Through these analyses and methodologies, the paper aims to provide insights and advancements in the field of time series clustering, particularly in terms of data reduction, efficiency, and clustering effectiveness.

The research from 2015 by Paparrizos et al. [79] develops a scalable domain-independent algorithm for time series clustering. The key focus is on creating a clustering algorithm that preserves the shapes of time series sequences and is invariant to scaling and shifting. The proposed k-Shape algorithm is a centroid-based clustering approach that aims to generate homogeneous and well-separated clusters. It utilizes an iterative refinement procedure similar to k-means but with significant differences in distance measure and centroid computation. Efficient steps are implemented in the algorithm, including cluster membership updates and centroid refinements, to achieve accurate clustering results. Through experimental evaluations and methodological advancements, the paper provides insights into effective clustering techniques for time series analysis.

The work by Guijo-Rubio et al. from 2021 [40] introduces a novel clustering methodology for time series data, referred to as the two-stage statistical segmentation-clustering time series procedure (TS3C). This methodology aims to improve the quality of clustering by first segmenting each time series into sub-sequences, extracting statistical features, and clustering these segments. Then, a second clustering stage is applied to the mapped time series to identify groups based on common patterns. The overall goal is to enhance the clustering process by exploiting similarities found in the segments of individual time series and improving the final clustering quality. This paper analyzes time series clustering by proposing a novel methodology, the two-stage statistical segmentation-clustering time series procedure (TS3C), which focuses on characterizing segment typologies within time series data. By incorporating segmentation (patterns discovery), feature extraction (segments variance, skewness, and autocorrelation coefficient), dimensionality reduction (mapping segments to statistical features), and a two-stage clustering process, this paper provides a comprehensive analysis of time series clustering that aims to enhance clustering quality and capture the similarities within segment typologies for improved clustering outcomes.

Bonifati et al., in their work from 2022 [9], introduce an interpretable and efficient end-to-end clustering system (Time2Feat) for multivariate time series (MTS) data. This system aims to provide users with insights into the clustering process while maintaining efficiency. The comprehensive evaluation of Time2Feat against state-of-the-art MTS clustering systems on various datasets is also a key objective of the paper. By incorporating interpretable features, human expertise, and efficient clustering techniques, the paper provides a detailed analysis of time series clustering, emphasizing the importance of interpretability and accuracy in real-world applications.

The works from 2022 and 2023 by Ji et al. [49][50] address the challenge of accurately classifying time series data based on intuitively interpretable features. The paper proposes a

novel Time Series Classification method based on Temporal Features (TSC-TF) that aims to generate temporal feature candidates, select important features using a random forest, and train a fully convolutional network for high accuracy in classification. Experimental validation on various datasets from the UCR Time Series Classification archive demonstrated the effectiveness of the proposed method in accurately classifying time series based on intuitive temporal features. These contributions highlight the significance of the proposed approach in addressing the challenges of time series classification based on interpretable features.

The works from Guijo-Rubio et al. [40] and Bonifati et al. [9] were chosen for the evolution patterns modeling and multivariate time series forecasting, respectively. The methodology from Guijo-Rubio et al. [40] fits the task of segmenting and clustering time series segments to identify generalized patterns. Bonifati et al. [9] specific focus on multivariate time series makes their work a suitable choice for the clustering task of this work. Additionally, both were reproducible and easily adaptable since the source code was made available.

2.3.3 Time Series Forecasting

Jones et al., in their work from 2009 [51], aim to study the temporal relationships between the demands for key resources in the emergency department (ED) and the inpatient hospital and to develop multivariate forecasting models. The study seeks to understand the dynamics of demand in the ED, develop models for forecasting ED census and critical resource demand, and explore the potential utility of multivariate forecasting models for decision support in real-time for on-call nurse staffing. The contributions of this paper include: providing insights into the temporal relationships between the demand and availability of emergency departments; developing multivariate forecasting models that offer more accurate forecasts compared to a univariate benchmark model; exploring the potential utility of multivariate forecasting models for decision support in real-time. These contributions enhance our understanding of demand dynamics in emergency departments and offer valuable insights for healthcare management and decision-making.

In their work from 2015, Kattan et al. [54] propose an unsupervised learning framework based on genetic programming (GP) for predicting the position of a particular target event defined by the user in an unseen time series. The framework aims to learn the behavior of the environment that generates the time series and use this knowledge to predict when the target event is likely to occur. The goal is to provide a method that does not require labeled data and can be applied to various domains such as stock markets, buyer-seller negotiations, or international market prices. The framework learns the behavior of the environment by analyzing historical time series vectors and using GP to evolve programs that distinguish different behaviors in the training data. By understanding the patterns that indicate the occurrence of target events, the framework can predict when these events are likely to occur in unseen time series data. GP is used to automatically build a library of candidate temporal features from historical time series vectors generated from the same environment. This approach allows the framework to capture the underlying patterns and behaviors in the data without the need for manual feature engineering. Overall, the paper provides a novel perspective on time series forecasting by emphasizing the importance of understanding the

generating environment's behavior and using this knowledge to predict specific events in time series data

The paper from 2018 by Chang et al. [16] presents a novel deep learning model called MTNet that addresses the challenges of multivariate time series forecasting. The goal is to improve the accuracy of time series predictions by capturing long-term dependencies and incorporating information from multiple variables in a way that is both effective and interpretable. The paper aims to demonstrate the effectiveness of MTNet through extensive experiments on benchmark datasets and compare its performance with state-of-the-art methods in both univariate and multivariate time series forecasting tasks. The paper analyzes time series forecasting by proposing a novel deep learning model, MTNet, specifically designed to address the challenges associated with multivariate time series data. MTNet consists of a memory component, three separate encoders, and an autoregressive component. These components work together to capture long-term dependencies and patterns in multivariate time series data. MTNet incorporates an attention mechanism that allows the model to focus on relevant segments of historical data when making predictions. This attention mechanism enhances the interpretability of the model by highlighting the importance of different parts of the input data. The paper analyzes the attention weights assigned by MTNet to different segments of historical data. By visualizing these attention weights, the model's ability to capture and utilize relevant information from the input data is assessed, providing insights into the forecasting process.

The work by Wan et al. from 2019 [99] discusses a novel deep learning model that can effectively capture long-term dependencies in multivariate time series data for accurate forecasting. The paper introduces the Multivariate Temporal Convolutional Network (M-TCN) model, which is specifically designed for multivariate time series forecasting. The model utilizes deep neural networks and convolutional architectures to capture long-term dependencies in the data. The performance of the M-TCN model is compared with traditional baseline models such as naive forecast, average approach forecast, and seasonal persistent forecast models. This comparison helps evaluate the effectiveness and superiority of the proposed model in time series forecasting tasks. The study utilizes the Walk-Forward Validation method to test the M-TCN model without updating it. This approach involves making predictions for a period of time and then comparing them with actual data to assess the model's forecasting accuracy and performance. The paper provides specific experimental details such as input lengths, batch size, loss function, optimization strategy (Adam), initial learning rate, and learning rate adjustments. These details contribute to the thorough analysis of the model's performance in time series forecasting tasks. By incorporating these methodologies and approaches, the paper conducts a comprehensive analysis of time series forecasting using the M-TCN model, demonstrating its efficiency and accuracy in capturing complex dependencies in multivariate time series data.

Cao et al., in their work from 2020 [11], develop a novel model, StemGNN (Spectral Temporal Graph Neural Network), that can effectively capture both intra-series temporal patterns and inter-series correlations in multivariate time series data. The goal is to leverage the benefits of Graph Fourier Transform (GFT) and Discrete Fourier Transform (DFT) to model time series data entirely in the spectral domain, enabling clearer patterns and more

effective predictions. StemGNN incorporates a carefully designed block that applies GFT to transfer structural multivariate inputs into spectral time series representations and DFT to transfer univariate time series into the frequency domain. By doing so, the spectral representations become easier to recognize by convolution and sequential modeling layers, leading to improved forecasting results. Additionally, StemGNN includes a latent correlation layer to automatically learn inter-series correlations, making it a general approach applicable to various multivariate time series forecasting tasks. The paper analyzes time series forecasting by proposing a novel approach, StemGNN, that leverages Graph Fourier Transform (GFT) and Discrete Fourier Transform (DFT) to capture inter-series correlations and temporal dependencies jointly in the spectral domain. Overall, the analysis of time series forecasting in this paper focuses on the development of a novel model that integrates spectral domain representations, automatic learning of inter-series correlations, and the application of GFT and DFT to enhance forecasting accuracy in multivariate time series data.

The paper by Du et al. from 2020 [28] introduces a novel multivariate time series multi-step forecasting model using an attention-based sequence-to-sequence learning structure. The goal is to effectively forecast multi-step time series values under different conditions by leveraging the encoder-decoder architecture with attention mechanisms. The paper introduces an encoder-decoder deep learning structure with a temporal attention mechanism to address the limitations of traditional methods. The model encodes hidden representations of multivariate time series data using Bi-LSTM and decodes them for multi-step forecasting. By incorporating a temporal attention layer between the encoder and decoder networks, the model can select relevant encoder hidden states across all time steps for more accurate forecasting. This mechanism enhances the model's representation ability of dynamic multivariate time series data. The analysis includes experiments on five multivariate time series datasets to evaluate the performance of the proposed model. The results demonstrate the effectiveness of the model in forecasting multi-step time series values under different conditions, showcasing its superiority over baseline methods.

In their work from 2020, Wu et al. [103] develop a general graph neural network framework specifically designed for multivariate time series data. The paper introduces a graph-based perspective for analyzing multivariate time series data. It views variables from multivariate time series as nodes in a graph interconnected through hidden dependency relationships. This perspective allows for the exploration of temporal trajectories while capturing interdependencies among time series variables. The paper identifies key challenges in existing approaches to time series forecasting using graph neural networks. These challenges include dealing with unknown graph structures and the need for simultaneous learning of the graph structure and the GNN for time series data. To overcome these challenges, the paper proposes a novel framework consisting of a graph learning layer, a graph convolution module, and a temporal convolution module. The graph learning layer extracts a sparse graph adjacency matrix adaptively based on the data, while the graph convolution module addresses spatial dependencies among variables. The temporal convolution module captures temporal patterns using modified 1D convolutions. The proposed framework allows for end-to-end learning, where all parameters are learnable through gradient descent. This approach enables the model to simultaneously model multivariate time series data and learn

the internal graph structure, addressing the challenge of graph learning and GNN learning. The paper presents experimental results showing that the proposed method outperforms state-of-the-art methods on benchmark datasets and achieves competitive performance on traffic datasets with structural information. This analysis demonstrates the effectiveness of the proposed framework in improving time series forecasting accuracy.

The work by Challu et al. from 2022 [15] introduces a novel forecasting model that addresses challenges in long-horizon forecasting. The N-HiTS model aims to improve forecasting accuracy by incorporating hierarchical interpolation and multi-rate data sampling techniques. By synchronizing input sampling rates with output interpolation scales and leveraging hierarchical structures, the model aims to enhance predictions for various frequency bands in time series data. The ultimate goal is to achieve state-of-the-art results on large-scale benchmark datasets commonly used in long-horizon forecasting research. The paper compares the performance of the N-HiTS model with several alternative models, including variations of N-HiTS with different components enabled or disabled, as well as existing models like N-BEATS. This comparison highlights the effectiveness of the proposed techniques in improving long-horizon forecasting accuracy.

The paper from 2023 by Chen et al. [18] aims to provide a novel approach to time series forecasting for cumulative data that effectively addresses monotonicity and irregularity issues. The paper starts by analyzing the monotonic increasing property of cumulative data in time series forecasting. It highlights the challenges posed by monotonicity, such as non-stationarity and large variances in the data, making the model training process challenging. To address the challenges of monotonicity, the paper proposes predicting the growth rate of cumulative data instead of exact values. By focusing on modeling the growth rate, the paper aims to ensure the monotonicity of predicted values during the inference stage. The paper also addresses the issue of irregularity in cumulative data, caused by errors like missing data or ”not a number” (NaN) entries. To mitigate irregularity challenges, the paper discusses incorporating time difference information into the model, although the uncertainty of errors poses a challenge. Based on the analysis of monotonic properties and irregularities, the paper proposes the Monotonic Ordinary neural Differential Equation (MODE) model within the framework of neural ordinary differential equations. The MODE model is designed to effectively capture the monotonicity and irregularity of cumulative data, providing a principled approach to time series forecasting. Extensive experiments are conducted to validate the effectiveness of the MODE model in simulation, offline, and online environments. The experimental results demonstrate the superiority of the MODE model in forecasting cumulative data, showcasing its ability to address monotonicity and irregularity challenges in time series forecasting.

The described previous work about multivariate time series forecasting presents many models that have been developed for that task. The methodology from Jones et al. [51] was chosen to be applied to this work as it described a footprint to follow rather than a specific forecasting model. Additionally, the neural network-based models were not adaptable to the available computational resources. Therefore, the approach from Jones et al. [51] allowed the creation of lighter models that still suited the aim of the forecasting task of the presented framework.

Chapter 3

Framework Design and Implementation

This chapter aims to present the methodology used to build the components of the framework introduced in this thesis. In section 3.1 an overview of this framework is given. Section 3.2.1 discusses the process of metrics selection through previous literature and the linking of these measures to the GitHub API[1]. The building of the dataset using these findings is addressed in section 3.2.2. Sections 3.3.1 and 3.3.2 illustrate the steps taken to build a model that maps the time series metrics data to generalized pattern curves and to merge time series into a unique representation. The clustering of open-source software projects based on their metrics evolution patterns is discussed in section 3.3.3, where the development of a related model is also shown. Section 3.3.4 describes the training and testing of multiple forecasting models for each metric in the identified project clusters.

3.1 Framework Overview

The framework presented in this thesis is a combination of time series methodologies [40][57][9][20][51] applied to open-source software evolution to fulfill several tasks: break points detection; evolution patterns modeling; metrics time series combination; repositories clustering; metrics values forecasting. The framework methodologies described in the following sections can be applied to a different set of metrics and used to the extent that researchers need them. Thus, the presented work can be taken as is or easily expanded by introducing more projects, a different set of metrics and additional methodologies. Overall, the framework pipeline is illustrated in figure 3.1 and summarized in the following steps:

1. **Metrics Data Collection**: Given an open-source project and a set of metrics, the historical data is collected from the GitHub API and stored in a database for easy retrieval.

[1] `https://docs.github.com/en/rest?apiVersion=2022-11-28`

2. **Metrics History Patterns Modeling**: The metrics history data are split into segments, which are then classified into related patterns. This step outputs a sequence of patterns per metric alongside the break points when a change in pattern occurs.

3. **Metrics Curves Merging**: The metrics patterns are merged into a single curve that provides a representation of the evolution of the open-source software project.

4. **Finding Similar Projects**: The analyzed project is assigned to one of the identified clusters to allow the finding of other similar projects. The clustering also allows the selection of the correct forecasting model, which is trained using data with similar evolution patterns.

5. **Evolution Forecasting**: With the forecasting models is possible to predict the future values of specific metrics based on the evolution of the other ones. The predicted time series can then be processed to obtain further break points and evolution patterns.

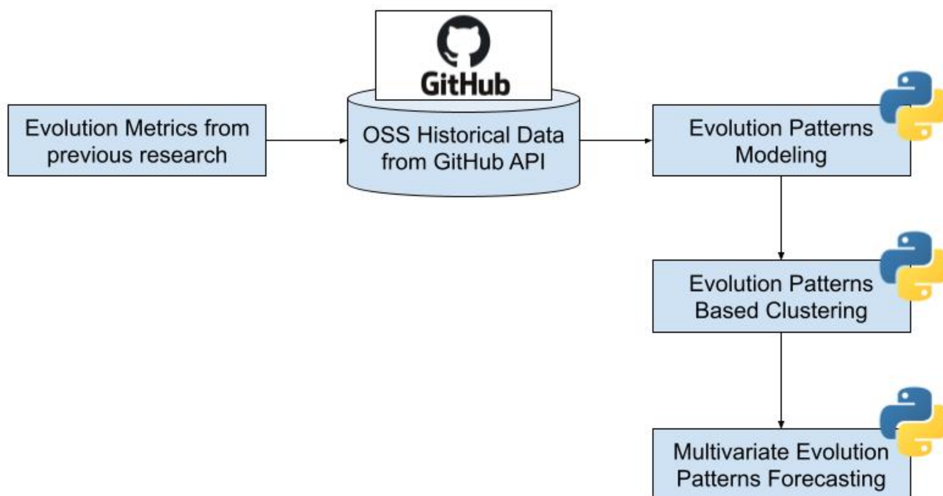

Figure 3.1: Framework pipeline steps. Source: Thesis author

Overall, the framework aims to support researchers in their studies of open-source software evolution. This is achieved by leveraging multivariate time series techniques to identify the evolution patterns of the projects' metrics to derive which factors and causes influence their growth.

3.2 Metrics Selection and Dataset

Section 3.2.1 illustrates the methodology adopted to select the metrics from previous literature, while section 3.2.2 shows how they were used to build the open-source software projects dataset.

3.2.1 Metrics Selection

The methodology adopted for the metrics selection process started with the gathering of relevant literature following the systematic review strategy discussed by Kitchenham et al. in their work from 2013 [60]. The definition of search keywords was the first action taken. In this regard, three sets of keywords were defined and potential pertinent synonyms were found. Table 3.1 shows the chosen main keywords and the related synonyms.

Main Keyword	**Context Synonyms**
Evolution	Maturity, Growth, Success
Open-source Software (OSS)	Open-source projects, Free software, Free Open-source Software (FOSS), Free/Libre and Open-source Software (FLOSS), Libre software

Table 3.1: Keywords and context related synonyms used for the first round of papers search

After the keywords selection, the selection of previous papers was carried out. Google Scholar [2] was used as the source to find papers and initial targets of time-range and minimum citations were set to 2010-2023 and 100, respectively. The goal was to identify at least 40 papers to analyze on further steps. The first iteration was carried out using the main keywords, which needed to appear both in the title and the abstract of potential papers. Synonyms were also used as well in the next iterations. This resulted in the choice of the following keywords, which contributed to the gathering of 50 papers in total: Evolution; Growth; Open-Source Software; Open-Source Projects.

The further iterations of the articles selection step focused on assessing that the set targets for minimum citations and time range would provide enough relevant papers. In this regard, it was noticed that the time range 2010-2023 returned less than 40 papers (25) that discussed the evolution of open-source software using tangible metrics. Therefore, the range was expanded to 2000-2023. This new setting resulted in 15 more relevant articles that could be analyzed. On top of this, the minimum number of citations set to 100 was too restrictive, especially for papers published in recent years. Thus, other iterations for the time range between 2000 and 2023 were run with minimum citations of 50 and without minimum citations at all for the papers between 2010 and 2023. This achieved the goal of gathering 40 papers to review for the next step of the systematic literature review. In the eventuality of missing this target, the iterations would have been repeated starting with alternative keywords.

[2] `https://scholar.google.com/`

Summarizing Questions
What are some of the most important factors that contribute to the evolution of open-source software projects, according to the authors of this paper?
What are some practical implications of the findings presented in this paper for individuals and organizations involved in open-source software development?
What is the used methodology?
Why are the results relevant?
Which metrics are used to define the evolution of an open-source project?

Table 3.2: Questions used to review and summarize a paper with the goal of guiding its quality-assessment score

Quality-Assessment Questions	**Weight**
Does the paper provide a set of factors and/or metrics to define the evolution of open-source projects?	0.4
Does the paper build upon relevant studies and/or research involving open-source contributors and users?	0.25
Are the provided factors/metrics tangible?	0.35

Table 3.3: Quality-assessment questions used to score a paper between 0 and 1. Papers with a score greater than 0.5 were selected

The articles inspection step started with the definition of the research questions, as discussed in the introduction. Based on the identified aims of the work, a set of questions that would support the papers review and summary were drafted as well as three quality-assessment questions that would provide an importance score of a paper. Each quality-assessment question was given a score of 1 (the paper addresses the topic in detail), 0.5 (the paper partially addresses the topic), or 0 (the paper does not address the topic at all). The three quality-assessment questions started with equal weights and the average of the three scores was assigned as the final score to an article. If the score was higher than 0.5, an article would be selected to be included in the literature review. The minimum target of 30 papers to be included in the final work was then established to guide the inspection and filtering process. The summarizing questions and quality-assessment questions and weights are listed in tables 3.2 and 3.3, respectively. A total of 31 papers were selected for the final work after this step. In the eventuality of missing this minimum target, the inspection step would have been run again on the excluded papers using additional questions. The analysis of the papers allowed the extraction of the factors used to define the evolution of open-source software projects. Tables 3.4 and 3.5 illustrate which factors are listed in the analyzed literature and provide a total count of how often they appear in the reviewed research.

The next step in the metrics selection process was the identification of tangible measures for each one of the identified factors. To fulfill the aim of studying the evolution patterns

Papers/Evolution Factors	**Licensing**	**Contributors**	**Subscribers**	**Footprint**	**Contributions**	**Best Practices**	**Releases**	**Issues**	**Size**	**Complexity**
Godfrey (2000)	•	•	•							
Godfrey (2002)		•	•	•						
Nakakoji (2002)		•	•		•					
Scacchi (2003)		•	•		•					
Paulson (2004)		•	•			•				
Aberdour (2007)		•	•			•				
Capiluppi (2007)	•	•	•		•		•			
Wang (2007)		•	•		•			•		•
Koch (2007)		•	•		•				•	
Nakagawa (2008)		•				•				
Xie (2009)		•	•			•	•	•		
Hotta (2010)	•	•			•	•	•			
Breivold (2010)					•			•	•	•
Karus (2011)				•	•					
Khan (2012)		•	•			•				
Capiluppi (2012)		•	•			•			•	•
Crowston (2012)		•	•		•					
Syeed (2013)		•	•	•		•				
Alenezi (2015)		•	•		•	•			•	•
Kumar (2020)	•	•	•		•	•				
Tandon (2020)		•	•		•		•	•		
Baabad (2020)		•	•		•	•	•	•		
Dawood (2020)		•	•		•	•				
Bogart (2021)		•	•	•	•	•				
Molnar (2021)		•	•		•	•	•	•		
Michelon (2022)	•	•	•		•	•				•
Kaur (2022)		•	•		•	•				
Zhang (2022)		•	•	•	•					
Chakroborti (2023)		•	•		•	•	•			
Jain (2023)		•	•		•					
Jallow (2024)	•	•	•		•	•				
Total	**6**	**29**	**27**	**5**	**22**	**18**	**7**	**6**	**4**	**5**

Table 3.4: Grouping of open-source software evolution factors cited by surveyed literature

Evolution Factor	Citations
Number of contributors	29
Number of subscribers	27
Number of contributions	22
Best practices adoption	18
Number of releases	7
Licensing	6
Number of issues	6
Footprint	5
Complexity	5
Size	4

Table 3.5: Summary of the open-source software evolution factors with their number of citations in the reviewed literature

of open-source software, the focus of this selection was directed toward metrics that are already represented as tangible time series data. This is because this type of information is efficiently retrievable and the data mining task can be automatized and scaled [61]. Non-tangible metrics can also be quantified as time series but were purposely excluded from this work to prioritize a more systematic data collection approach. The chosen source of data was the online version control management system GitHub [69][68][96], which hosts more than 420 million repositories[3] of which more than 100 million are public[4], at the time of writing. Leveraging on the GitHub REST API[5], it is possible to access historical data from any open-source repository hosted on GitHub. Therefore, the gathered factors from the literature were mapped to the following metrics from the API:

- **Number of contributors:** The amount of developers that contributed to the source code over time is gathered from the list of contributors provided by the REST API[6]. The date of the first commit made is used as a time reference for the time series data. The number of contributors over time is represented by a monotonically non-decreasing curve.

- **Number of subscribers:** GitHub interprets subscribers as the number of users that marked a repository with a star[7,8][89], who are referred to as stargazers. Therefore, the number of stargazers assigned over time has been gathered from the related API

[3] https://github.com/about
[4] https://github.com/search?q=is%3Apublic&type=repositories
[5] https://docs.github.com/en/rest?apiVersion=2022-11-28
[6] https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28#list-repository-contributors
[7] https://github.com/holoviz/param/issues/273
[8] https://docs.github.com/en/rest/activity/watching?apiVersion=2022-11-28

endpoint[9]. The number of stargazers over time is represented by a monotonically non-decreasing curve.

- **Number of contributions:** The amount of activity that developers contribute towards an open-source project is measured by the number of commits and pull requests to the main branch [61][14][7]. Therefore, the related endpoints of the GitHub API are used to gather the commits over time[10] and merged pull requests over time[11] to the main branch. Both commits over time and pull requests over time are represented by monotonically non-decreasing functions.

- **Best practices adoption:** As this factor is quite abstract to measure, metrics related to the presence of a continuous integration and continuous deployment (CI/CD) pipeline were taken into account [14]. Therefore, the number of successful workflows[12] and successful deployments to a production environment[13] over time were retrieved from the GitHub API. Both time series are represented by monotonically non-decreasing curves.

- **Number of releases:** The number of releases over time is taken directly from the related GitHub API endpoint[14]. The time series is represented by a monotonically non-decreasing curve.

- **Licensing:** The presence of a license and its eventual type is not represented by a tangible measure over time by the GitHub API. Therefore, this factor has been omitted from the data collection.

- **Number of issues:** The GitHub API provides the list of issues over time that are still open at the time of the request[15]. This way, it is possible to get sense of how many issues have accumulated since the start of the project. The number of open issues over time is represented by a monotonically non-decreasing curve.

- **Footprint:** The footprint of a project is also an abstract factor since it can be interpreted in different ways. By following previous research [82][33][68][96][21], the project's footprint has been linked to the size of the network of projects that has been created using the project itself as a starting point, which can be measured by the

[9] https://docs.github.com/en/rest/activity/starring?apiVersion=2022-11-28#list-stargazers

[10] https://docs.github.com/en/rest/commits/commits?apiVersion=2022-11-28#list-commits

[11] https://docs.github.com/en/rest/pulls/pulls?apiVersion=2022-11-28#list-pull-requests

[12] https://docs.github.com/en/rest/actions/workflows?apiVersion=2022-11-28#list-repository-workflows

[13] https://docs.github.com/en/rest/deployments/deployments?apiVersion=2022-11-28#list-deployments

[14] https://docs.github.com/en/rest/releases/releases?apiVersion=2022-11-28#list-releases

[15] https://docs.github.com/en/rest/issues/issues?apiVersion=2022-11-28#list-repository-issues

number of forks[16]. The number of forks over time is represented by a monotonic non-decreasing curve.

- **Complexity:** The GitHub API does not provide a metric that can be used to measure the complexity of a repository over time as parameters such as the number of dependencies and the number of programming languages are only available with their most recent snapshot. Therefore, this factor has been omitted from the data collection.

- **Size:** The size of a project can be estimated by adding and subtracting the addition and deletion changes from different commits. However, the GitHub API does not return such information from the commits listing endpoint and it is required to request data for each commit. This task is quite cumbersome and it results in a volume of API requests that exhaust the permitted limit of 5000 requests per hour[17]. As repositories differ highly in the amount of commits, the execution time of this specific metric gathering is not scalable. On top of this, the GitHub API endpoint that returns the weekly additions and deletions is limited to repositories with less than 10000 commits[18], which would limit the scope of this research significantly. Therefore, a down-sampling approach has been taken to gather this metric data efficiently. The repository commits were grouped by month and one commit was taken for each month. The stored results from the GitHub API endpoint[19] included the number of additions, deletions and total changes related to each requested commit. Additionally, the difference of additions and deletions has been stored as a project size change reference. The cumulative number of changes is represented by a monotonic non-decreasing curve, while the size change is represented by a non-monotonic curve. The cumulative number of changes has been used for further computations to keep it consistent with the other metrics time series.

The mapping of the retrieved factors from previous literature to the GitHub API metrics is summarized in table 3.6. Overall, the identified factors and the related tangible metrics are used to gather time series data for different open-source GitHub projects. The process of building the dataset is discussed in the next section (3.2.2).

3.2.2 Building the Dataset

As discussed in section 3.2.1, GitHub is home to more than 100 million open-source projects. Therefore, the GitHub API is used to retrieve the metrics data necessary for the realization of this research. However, consideration of which projects to use to build the dataset is required since there is the risk of selecting projects that are not diverse enough or that do not have enough data to offer.

[16] https://docs.github.com/en/rest/repos/forks?apiVersion=2022-11-28#list-forks

[17] https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api?apiVersion=2022-11-28#primary-rate-limit-for-authenticated-users

[18] https://docs.github.com/en/rest/metrics/statistics?apiVersion=2022-11-28#get-the-weekly-commit-activity

[19] https://docs.github.com/en/rest/commits/commits?apiVersion=2022-11-28#get-a-commit

Evolution Factor	GitHub API Metric
Number of contributors	Contributors
Number of subscribers	Stargazers
Number of contributions	Commits, Pull requests
Best practices adoption	Workflows, Deployments
Number of releases	Releases
Licensing	-
Number of issues	Open issues
Footprint	Forks
Complexity	-
Size	Commit changes

Table 3.6: Mapping of the open-source software evolution factors to the available metrics from the GitHub API

In their work from 2023, Chakroborti et al. [14] needed to build a dataset of open-source software projects to inspect their release practices. As they used both the Software Heritage Graph Dataset (SWHGD)[20] and the GitHub API to find common development patterns, a large amount of data mining was required. The applied repository filtering was based on: presence of development activities; opening and closing of issues; project boards management. Thus, the final list of candidates from Chakroborti et al. was chosen as a starting point to build the time series dataset of the metrics discussed in section 3.2.1. All the 1800 repositories in the list have been processed and their details have been requested from the GitHub API using the endpoints listed in table 3.7. It occurred that some of the projects in the starting list were not available on GitHub anymore, which led to their exclusion from the data collection. The GitHub data mining process resulted in a dataset of 1328 repositories with related evolution metrics time series data. The dataset created is openly available at the following URL: `https://huggingface.co/datasets/MattiaBonfanti-CS/IN5000-MB-TUD-Dataset-MongoDB`.

In terms of data storage, MongoDB[21] was chosen as the database due to its flexibility[52][70] since the data schema evolved through the research. The repositories' general information, which summarizes the main properties of an open-source project, was stored in a collection named *repositories_data* and assigned a unique identifier. This collection acts as a central point for data retrieval as each metric time series data has been stored in dedicated collections to avoid exceeding the MongoDB object size limit of 16MB[22]. Each object in the metrics collections has a *repository_id* value that references the *repositories_data* collection to ease data retrieval. The database schema overview is illustrated in appendix A.

The creation of the dataset of 1328 open-source software projects metrics data represents one of the contributions of the current work as it can be updated, expanded and re-used

[20] `https://docs.softwareheritage.org/devel/swh-dataset/graph/`

[21] `https://www.mongodb.com/`

[22] `https://www.mongodb.com/docs/v5.2/reference/limits/`

Evolution Factor	GH API Metric	GH API Endpoint
Number of contributors	Contributors	/repos/{owner}/{repo}/contributors
Number of subscribers	Stargazers	/repos/{owner}/{repo}/stargazers
Number of contributions	Commits	/repos/{owner}/{repo}/commits
	Pull requests	/repos/{owner}/{repo}/pulls
Best practices adoption	Workflows	/repos/{owner}/{repo}/actions/workflows
	Deployments	/repos/{owner}/{repo}/deployments
Number of releases	Releases	/repos/{owner}/{repo}/releases
Number of issues	Open issues	/repos/{owner}/{repo}/issues
Footprint	Forks	/repos/{owner}/{repo}/forks
Size	Commit changes	/repos/{owner}/{repo}/commit/{sha}
General Information		/repos/{owner}/{repo}

Table 3.7: Evolution factors with the corresponding GitHub API metrics and retrieval endpoints

beyond the scope discussed in this thesis. Figure 3.2 illustrates an example of a collected metric time series data of an open-source software project.

3.3 Multivariate Time Series Models

This section aims to show the methodology used to apply the multivariate time series techniques from other domains (finance and healthcare) [40][9][20][51] to the study of open-source software evolution. The patterns modeling is presented in section 3.3.1 followed by the time series merging in section 3.3.2. The multivariate time series clustering is discussed in section 3.3.3. Finally, the multivariate time series forecasting is described in 3.3.4.

3.3.1 Evolution Patterns Modeling

This step of the methodology describes how time series data, related to open-source software metrics, can be used to generate general evolution patterns. This results in an algorithm that can split time series data into segments, which are then clustered to produce generalized evolution pattern curves. A classifier model is also trained to predict the patterns sequence of new time series data. The identification of the pattern curves follows the first step of the two-stage methodology proposed by Guijo-Rubio et al. in their work from 2020 [40], where they discuss a novel time series clustering approach based on characterization of segment typologies. The part of this work that was implemented in the current research consists of applying time series segmentation via efficient polynomial approximation. Next, the heterogeneous segments are projected into feature vectors of equal length, to reduce the dimensionality of the original data and have the same length for each mapped segment. Hierarchical clustering is then applied to group the segments to recognize similar patterns. Finally, the clustering results are used to train a K-nearest neighbors (KNN) classifier model

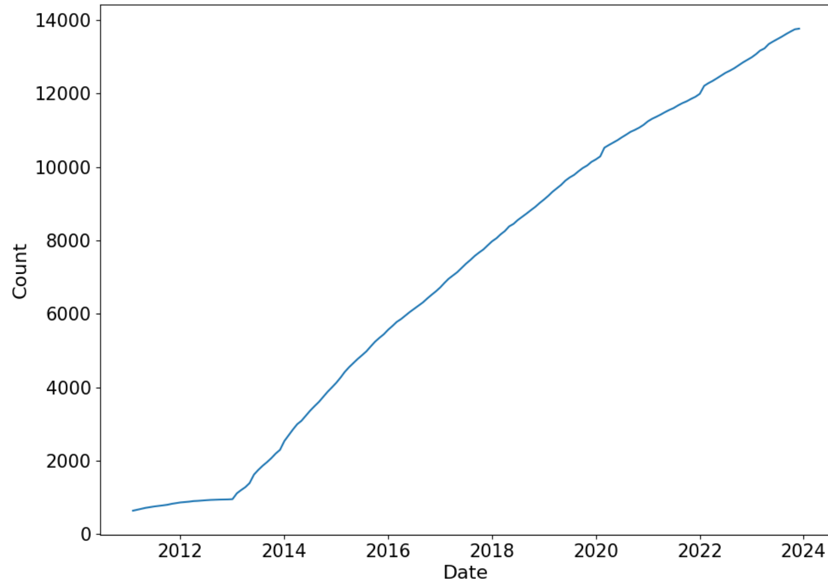

Figure 3.2: Time series data of the cumulative stargazers for the saltstack/salt project. Source: Thesis author

that can be used to identify the patterns of new time series data. The following definitions are applied to the key terms of this section:

- **Time Series**: A series of values over time $T = ((t_1, v_1), ..., (t_N, v_N))$ with t_i, v_i as the pair of registered value v_i at the related time t_i and N as the length of the data series.
- **Time Series Segment**: A set of values and times from a time series T between two time points. It is defined as $T'_{ij} = (t_i, v_i), ..., (t_j, v_j))$, with $T' \subseteq T$.
- **Evolution/Growth Pattern**: A representation of a time series segment as an approximated polynomial curve to establish its growth trend. Given a time series segment T' and a polynomial approximation function F, the evolution pattern is defined as $E = (e_0, e_1, ..., e_m)$, with e_i as a polynomial coefficient and m as the degree of the polynomial.
- **Evolution/Growth Patterns Sequence**: A series of pattern curves that approximate a sequence of time series segments, defined as $G = (E_1, ..., E_K)$ with K the number of time series segments.

Given a time series of length n, a segmentation problem consists in finding m segments defined by a set of $m-1$ breaking points. This has been achieved by using the window-

based change point detection functionality of the Python package *ruptures*[23] [95]. The window-based change point detection algorithm uses two windows that slide along the data series and measure the discrepancy of the signals' statistical properties within each window range. Given a cost function $c(\cdot)$, a discrepancy $d(\cdot,\cdot)$ is computed as follows:

$$d(y_{u..v}, y_{v..w}) = c(y_{u..w}) - c(y_{u..v}) - c(y_{v..w})$$

with $y_{a..b}$ as the time series windowed data and $u < v < w$ the windows indexes. The discrepancy is the cost gain of splitting the signal interval $y_{u..w}$ at index v. If the sliding windows $u..v$ and $u..w$ both overlap with a segment, their statistical properties will be similar, thus their discrepancy will be low and no break point will be introduced. On the other hand, if the two windows overlap with different segments, their discrepancy will be significantly higher and a break point will be introduced. The discrepancy can be represented as a curve for all indexes t between $w/2$ and $n - w/2$, with w as the window length and n the time series length: $(t, d(y_{t-w/2..t}, y_{t..t+w/2})$. A sequential peak search is performed on this curve to detect the break points. In terms of cost function, the least squared deviation (L2), which detects mean shifts in a data series, has been applied. Considering $\{y_t\}_t$ a time series within an interval I and $\bar{y}$ the mean value of the data in the interval, the cost L2 function is defined as:

$$c(y_I) = \sum_{t \in I} \|y_t - \bar{y}\|_2^2$$

Overall, this window-based segmentation approach is well suited for this part of the methodology due to its low time complexity of $O(nw)$ (w the window length, n the time series length) and to the fact that it works even when the number of break points is not known beforehand. This is important as the built dataset includes time series of 10 heterogeneous metrics for more than 1300 projects. Additionally, this method adheres to the statistical approximation-based segmentation used by Guijo-Rubio et al. in their work from 2020 [40]. As the input time series data is grouped by month, a window size of 12 months has been used to identify the segments. Figure 3.3 illustrates the segmentation of metric time series data of an open-source software project obtained by following the explained procedure.

After the segmentation process, each segment is scaled to a $[0, 1]$ range to make the pattern identification independent from the metric values the segment is extracted from. This allows the isolation of the growth patterns of each time series from all the collected repository data and produces generalized trend curves. After this, the scaled segments are projected to l-dimensional arrays to allow their clustering. The mapping to the same dimensional space is performed by extrapolating the statistical properties of each segment using the Python *tsfresh*[24] [19]. The following statistical features are computed:

1. **Polynomial coefficients**: Least squares approximation of the segment data sequence, which provides the coefficients of a third-degree polynomial [32].

[23] `https://centre-borelli.github.io/ruptures-docs/`
[24] `https://tsfresh.readthedocs.io/en/latest/`

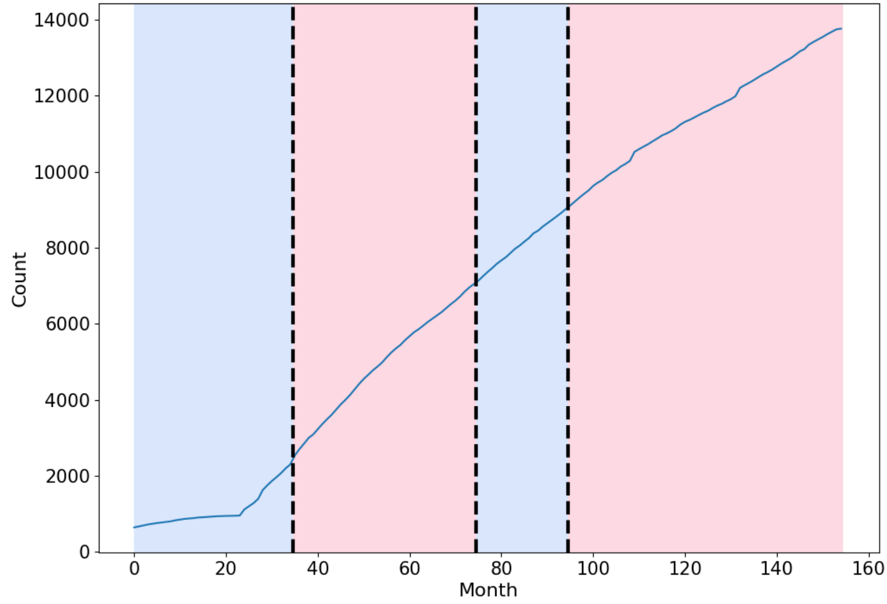

Figure 3.3: Stargazers time series segmentation for the *saltstack/salt* project. Source: Thesis author

2. **Variance**: Measure of the variability of the segment data.

$$S_s^2 = \frac{\sum_{i=1}^{n}(y_i - \overline{y_s})^2}{n-1}$$

with y_i as the values of the segment s of length n and $\overline{y_s}$ as their average.

3. **Skewness**: The asymmetry of the segment data distribution with respect to the mean value.

$$\gamma_s = \frac{\sum_{i=1}^{n}(y_i - \overline{y_s})^3}{(n-1)\cdot\hat{\sigma}_s^3}$$

with $\hat{\sigma}_s$ as the standard deviation of the segment s of length n.

4. **Autocorrelation coefficient**: Measure of the correlation between the data in the segment.

$$AC_s = \frac{\sum_{i=1}^{n}(y_i - \bar{y}_s)\cdot(y_{i+1} - \bar{y}_s)}{S_s^2}$$

With the aid of the statistical features listed above, it is now possible to map the segments to a l-dimensional space, where $l = c + f$ with c as the number of polynomial coefficients and f as the number of statistical properties. Thus, the segment mapping is defined as follows.

$$v_s = (c_0, c_1, c_2, c_3, S_s^2, \gamma_s, AC_s)$$

Now that all the segments are translated to the same dimensional space, it is possible to cluster them. The clustering technique used is hierarchical clustering with Ward distance as the similarity measure. The overall time series segmentation and segments clustering process is illustrated in figure 3.4. Finally, by leveraging on the results of the hierarchical clustering it is now possible to train a KNN classifier that can be used to assign the segments of an input time series to one of the clusters. The results of this methodology are shown in chapter 4.1.1.

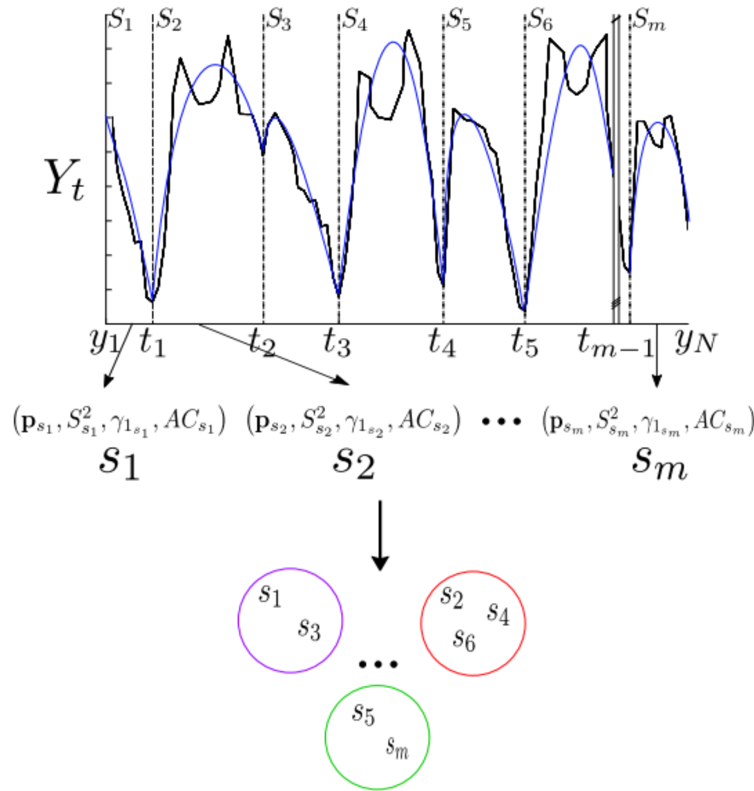

Figure 3.4: Time series segmentation and segments clustering process overview. Source: Guijo-Rubio et al., 2020 [40]

3.3.2 Time Series Merging

Now that a generalization technique for time series segments has been provided (3.3.1), it is important to convey all the metrics information to a single representation of a project's

status. This will allow researchers to have an efficient way to assess the evolution of a project before digging into the specific factors [57][67]. To achieve this, the metrics time series must be combined to show a unique signal that still reflects the previously identified break points of each metric.

The merging of metrics time series into a single representation follows the algorithm proposed by Keogh et al. from 1998 [57], where a merge operator enables the combination of information from two or more sequences, based on how many times it is applied. The output of this algorithm will be a sequence that represents a compromise between the two input sequences based on the relevance associated with each of their segments. To focus on the growth patterns, the metrics time series are scaled between 0 and 1, as they were during the segment extraction process. The break points of each metrics time series are merged into a single ordered set and used as a reference to compute the combined signal. This way the break points identified earlier are preserved in the final representation. The procedure is illustrated in algorithm 1 and figure 3.5. A combination example of the time series is shown in figure 3.6.

Algorithm 1 Merging Time Series Algorithm

Require: ts_1, ts_2, bp_1, bp_2
Ensure: $||ts_1|| = ||ts_2||; ts_1, ts_2 \in [0,1]$
  $BP = bp_1 \cup bp_2$
  for $i \leftarrow 1, ||BP||$ **do**
    $seg_1 \leftarrow (ts_1[i-1], ts_1[i])$
    $seg_2 \leftarrow (ts_2[i-1], ts_2[i])$
    $seg_avg \leftarrow (seg_1 + seg_2)/2$
    $pattern_1 \leftarrow get_pattern(seg_1)$
    $pattern_2 \leftarrow get_pattern(seg_2)$
    $weight_1 \leftarrow get_pattern_weight(pattern_1)$
    $weight_2 \leftarrow get_pattern_weight(pattern_2)$
    $weight_avg \leftarrow (weight_1 + weight_2)/2$
    $out_{ts}[i] \leftarrow seg_avg * weight_avg$
  end for
  $out_{ts} \leftarrow normalize(out_{ts})$
  return out_{ts}

Overall, the discussed methodology step allows an efficient representation of the historical data of a given open-source software. The resulting time series curve can then be mapped to a sequence of patterns as discussed in chapter (3.3.1).

3.3.3 Multivariate Time Series Clustering

The findings from chapter 3.3.1 allow the modeling of time series data as a sequence of labeled patterns based on the polynomial approximation and statistical features of their segments. Each open-source software metrics data is now represented homogeneously and can be used to define further generalizations on the evolution of such projects. This step of

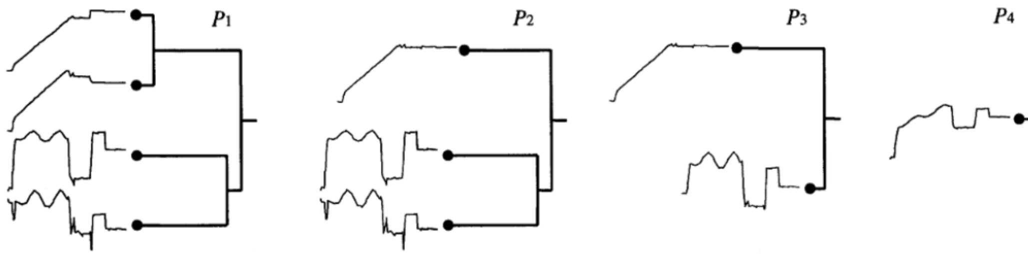

Figure 3.5: Time series merging procedure. Source: Keogh et al., 1998[57]

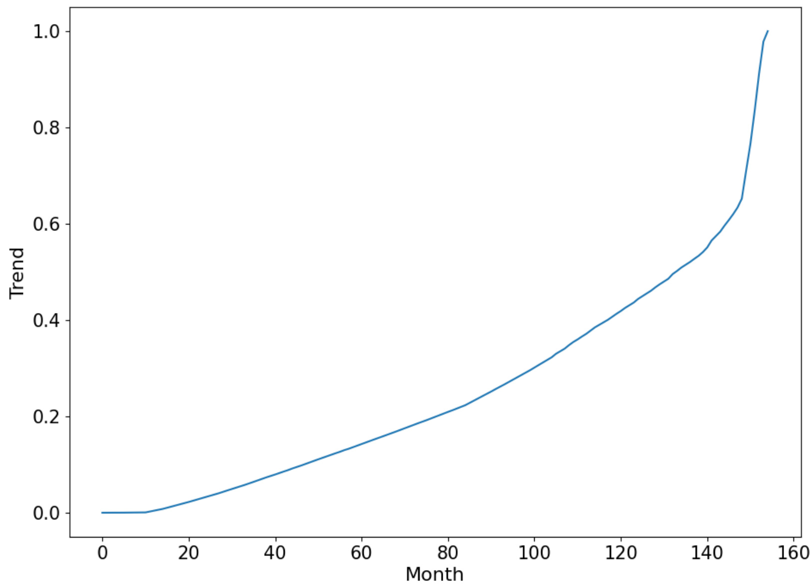

Figure 3.6: Merged metrics time series for the saltstack/salt project. Source: Thesis author

the methodology aims to categorize the growth behavior of open-source software projects by grouping them following their history patterns and their metrics correlation. This results in a clustering model that can identify similar open-source projects based on the described criteria.

The open-source software projects clustering procedure is based on the work by Bonifati et al. from 2022 [9], where they propose Time2Feat, a system for multivariate time series clustering relying on an end-to-end semi-supervised feature-based pipeline. In the specific case of this research, the repository data is used as a starting point. This historical data is presented as a set of heterogeneous time series, which need to be handled properly

to allocate them into clusters. The challenge is introduced by the fact that each repository's evolution data is characterized by several metrics time series. Therefore, the chosen clustering technique requires strong scalability to account for the multivariate nature of the repositories dataset. The approach used by the Time2Feat system relies on translating each multivariate data point to a vector composed of the intra-signal features and inter-signal features of the different time series. This way, the nature of each independent signal is included as well as the relationship between each signal pair. The elements of this multivariate clustering pipeline are defined as follows:

- **Multivariate Time Series**: A set of univariate time series $M = (u_1,...,u_S)$, where S is the number of such series and $u_j = (t_1,...,t_N)$ is a time series of length N. Therefore, a multivariate time series can be represented as a matrix $\mathbb{R}^{NxS}$ with the series as columns.

- **Multivariate Time Series Dataset**: A dataset D of V multivariate time series, defined as $D = (M_1,...,M_V)$ and represented as a tensor $\mathbb{R}^{VxNxS}$.

- **Intra-signal Features**: A set of statistical features for each independent time series in a multivariate time series $M = (u_1,...,u_S)$. Given a set of feature extraction functions $F = (f_1,..., f_F)$, the intra-signal features set is define as $F_{intra} = (e_{11},...,e_{1F},...e_{S1},...e_{SF})$ with $e_{ij} = f_j(u_i)$.

- **Inter-signal Features**: A set of correlation features between each pair of time series in a multivariate time series $M = (u_1,...,u_S)$. Given a set of correlation extraction functions $G = (g_1,...,g_G)$, the inter-signal features set is defined as $F_{inter} = (e_{(1,2)1},...,e_{(1,S)G},...e_{(S-1,S)G})$ with $e_{(i,j)k} = g_k(u_i,u_j)$.

- **Multivariate Time Series Clustering**: Definition of a surjective function $m : D \to C$ that maps each multivariate time series $M \in D$ into a cluster $k \in C$, with D the multivariate time series dataset and C a set of clusters.

The first step of applying the Bonifati et al. [9] methodology to the repositories dataset is the extraction of intra-signal and inter-signal features from the multivariate time series entries. For the intra-signal features, the evolution patterns modeling from chapter 3.3.1 is used as it provides a statistical approximation of the different metrics time series segments. Each metric signal can now be represented as a sequence of growth patterns, each one characterized by its polynomial coefficients, variance, skewness and autocorrelation values. The inter-signal features of a single metric are defined as the average of the statistical features of its evolution patterns. This approach is in line with what Time2Feat adopts as it uses the *tsfresh*[25][19] Python package to compute the same properties as already performed in chapter 3.3.1.

The inter-signal features extraction is based on measuring the similarity between each pair of metrics time series for a repository. The Time2Feat system proposes several distance

[25]`https://tsfresh.readthedocs.io/en/latest/`

measures (e.g. Correlation, Euclidean), which fit well the general study of time series similarity but present some challenges as the focus of this step is on the evolution patterns of each metric signal. These shortcomings are mainly because these distances are based on a point-to-point comparison and are affected by signals with different lengths, frequencies and amplitude [105][58]. Therefore, as repositories time series data are quite heterogeneous, a uniform representation is needed to properly compare the growth patterns of the different metrics. To achieve this, the pattern distance introduced by Chung et al. in their work from 2004 [20], which analyzed stock markets time series, provides a suitable solution as it overcomes the problems of pattern mismatch and works well with time series of different resolutions. This follows the choice of previous works [39][87][88] to rely on time series techniques applied to financial markets as the research on this topic is more well-developed.

To calculate the pattern distance between two time series, they both need to be converted to a piece-wise linear representation first, which represents a sequence of trends rather than raw values. The modeling consists of mapping time series values to a sequence of 0, 1, or -1 based on the growth between two points. This allows the translation of time series data into a trend sequence $S_t = \{(m_1,t_1),...,(m_N,t_N)\}$, with $m_i \in \{0,1,-1\}$ as the trend value and t_i as the end time point of the related trend. In particular:

- 0 when the value remained constant (continuation).
- 1 when the value increased (uptrend).
- -1 when the value decreased (downtrend).

Therefore, given two time series and the two respectively mapped trend sequences $S_1 = \{(m_{11},t_{11}),...,(m_{1N},t_{1N})\}$ and $S_2 = \{(m_{21},t_{21}),...,(m_{2N},t_{2N})\}$, with $t_{1i} = t_{2i}$ for $i = 1,...,N$, the pattern distance between S_1 and S_2 is defined as:

$$D_p(S_1,S_2) = \frac{1}{t_{1N}} \sum_{i=1}^{N} (t_{1i} - t_{1(i-1)})|m_{1i} - m_{2i}|$$

The value of the pattern distance D_p will be between 0 and 1 and smaller values will determine higher similarity in the two trend sequences. This distance is calculated for all pairs of time series in each repository in the dataset.

Finally, repository clustering was performed using the hierarchical clustering technique with Ward distance as a similarity measure. Leveraging on the results of the hierarchical clustering, it is now possible to train a KNN classifier that can be used to assign an open-source project to one of the defined clusters by using the intra-feature and inter-feature representation of its metrics data as input.

Overall, the discussed methodology step allows the clustering of open-source software projects based on their evolution patterns representation extracted following the techniques from chapter 3.3.1. The developed classifier can be used to identify similar projects and will serve as a starting point for the next step of the overall methodology, which handles the multivariate metrics evolution patterns forecasting. The results of this methodology are shown in chapter 4.1.1.

3.3.4 Multivariate Time Series Forecasting

To support the analysis of open-source software evolution, researchers should understand how the different metrics can evolve based on their history. Furthermore, the changes related to one metric can impact the evolution of other ones as well. Thus, multivariate time series forecasting is used to predict the evolution of the different metrics of an open-source software project. This last step of the methodology aims to provide a set of models (one per metric) for each repository cluster that can be used for the following purposes:

1. Predict the next patterns of open-source metrics based on their history data as they are.
2. Assess future evolution of metrics based on manually introduced patterns as assumed forecasts.
3. Replay and assess the evolution history of metrics based on manually introduced patterns at some point in the past.

The forecasting models tailored for different repositories clusters (3.3.3) enhance the ability to analyze open-source software evolution with the additional aid of the related evolution patterns (3.3.1), which allow the shaping of the metrics history to evaluate the impact of past events and predict future ones.

The application of multivariate time series forecasting in open-source software evolution is inspired by the research by Jones et al. from 2008 [51], where they apply this technique to predict the demand in emergency departments for hospitals. Although from a different domain, this work serves as a starting point as its methodology relies on studying the temporal relationships between the demands for key resources in the emergency department and the inpatient hospital. The relationship between the evolution of these different factors provides the basis for the multivariate forecasting model. The data used to train and evaluate this model was collected from three different hospitals and sampled by the hour for a specific year. The multivariate forecasting model produced by this research performed better than the univariate ones it was compared to. The described methodology gives the necessary steps to guide the development of multivariate forecasting models in the field of open-source software development.

The metrics time series data have been prepared for the models training by being sampled by month and being scaled between 0 and 1, as the patterns are the matter of interest here rather than the values themselves. Each metric has then been selected as the model target, in turn, to produce a forecasting model per metric, using the other ones as training data. The *mlforecast*[26] Python library [77], has been used to train the forecasting models. The advantage of this framework is that it already provides a series of machine learning models suitable for time series forecasting. Among these options, XGBoost [17] was chosen as the base model for the training task due to its speed, scalability and good performance [78][107][98].

[26] https://github.com/Nixtla/mlforecast

The described procedure resulted in the development of a set of forecasting models, which are divided evenly among the identified clusters and each represents one of the 10 metrics associated with the evolution of open-source software projects. These models can be used to predict future patterns for the metrics and to infer hypothetical scenarios by manipulating the input data with different pattern types. Overall, the discussed methodology step finalizes the elements that compose the support framework. The results of this methodology are shown in chapters 4.1.2 and 4.1.3.

Chapter 4

Results and Evaluation

This chapter aims to provide an answer to the research questions, which are discussed in each one of the following sections: 4.1 for RQ1; 4.2 for RQ2; 4.3 for RQ3. Each section describes the evaluation approach followed and illustrates the results obtained.

4.1 RQ1: What Insights can be Derived from the Framework?

This research question aims to show which useful information researchers can retrieve using the framework methodologies described in chapter 3 to gain a better understanding of open-source software evolution. The results from the patterns modeling and multivariate time series clustering are discussed in section 4.1.1. Additionally, in section 4.1.2, an assessment is conducted about how well the forecasting models fit the time series data and can predict evolution patterns for open-source projects metrics. Finally, the influence that metrics have on each others' growth is analyzed and the most influential evolution metrics are listed in section 4.1.3.

4.1.1 RQ1.1: What are the Generalized Patterns and Clusters of Open-Source Software Evolution?

The goal of this research question is to discuss the results obtained from the patterns modeling from chapter 3.3.1 and the multivariate time series clustering from chapter 3.3.3. First, the generated generalized patterns are discussed and an experiment is conducted to check if the results are consistent with smaller input sizes. Following that, the obtained repositories clusters are illustrated.

The methodology described in chapter 3.3.1 resulted in 3 clusters of segments grouped by the similarity of their polynomial approximation and statistical features. From this, it is possible to generalize the 3 curves that represent the segments' behavior in the time series data of the analyzed metrics. The average coefficients of the clustered segments are used to define the following third-degree polynomial equations:

1. **Steep growth**:

$$y = 0.03x^3 + 0.043x^2 - 0.305x + 1.725 + C$$

2. **Shallow growth**:

$$y = 0.012x^3 - 0.017x^2 + 0.027x - 0.011 + C$$

3. **Plateau**:

$$y = 0.0001x^3 + 0.001x^2 - 0.002x + 0.002 + C$$

The defined equations are illustrated by the curves in figure 4.1 and can be used to approximate the sequence of segments in the collected time series data. By providing a generalized representation of the metrics growth over time it is then possible to analyze their correlation and to group open-source projects with similar evolution patterns.

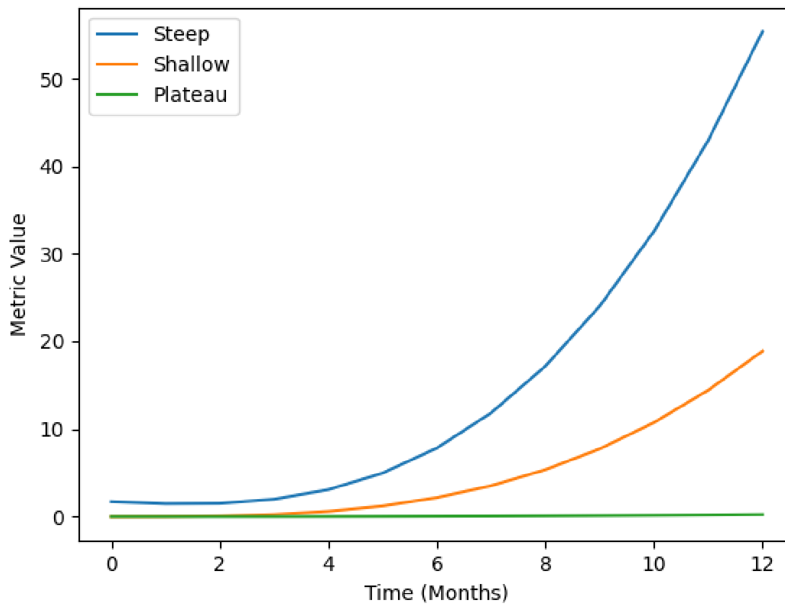

Figure 4.1: Generalized time series segments growth patterns. Source: Thesis author

The results from the experiments run with smaller input sizes still produced 3 general curves with similar behaviors to the ones obtained using the full dataset. This can be seen in table 4.1, where the polynomial coefficients are in line across the different iterations. The curve plots are shown in appendix B.

The multivariate time series clustering discussed in chapter 3.3.3 produced 3 clusters to which repositories are assigned. The dendrogram that resulted from the hierarchical clustering is illustrated in figure 4.2. The average metrics evolution patterns for each cluster are shown in figures 4.3, 4.4 and 4.5. From the plots, it can be seen how the projects in each cluster differ in terms of their evolution related to the metrics scale over time and their correlation. The plot for *Cluster_0* presents a growth for all the metrics, with some being faster than others. It is expected that the projects in this cluster mostly have continuous growth in their metrics values and that the evolution of one metric affects the others as well.

	Steep				Shallow				Plateau			
Input Size	x^3	x^2	x^1	x^0	x^3	x^2	x^1	x^0	x^3	x^2	x^1	x^0
1328 (Baseline)	0.03	0.043	-0.305	1.725	0.012	-0.017	0.027	-0.011	0.0001	0.001	-0.002	0.002
664	0.04	0.037	-0.139	0.289	0.01	-0.01	0.015	-0.004	0	0	0	0
332	0.04	0.059	-0.242	0.499	0.015	-0.013	0.02	-0.004	0	0	0	0
166	0.06	0.049	-0.318	0.887	0.02	-0.016	0.024	-0.005	0	0	0	0
83	0.08	0.022	-0.119	0.439	0.02	-0.02	0.034	-0.01	0	0	0	0
41	0.07	0.03	0.01	-0.06	0.023	-0.023	0.033	-0.01	0	0	0	0

Table 4.1: Generalized patterns curves coefficients for different input size, with full dataset of 1328 repositories as baseline

In total, 141 open-source projects were assigned to this cluster and it is assumed that the *Steep* and *Shallow* patterns are the most common in the metrics evolution. From the plot of *Cluster_1*, it can be seen that the metric growth is slower than in *Cluster_0* and that a few of them hardly evolve. The metrics growth correlation is expected to be limited, where the evolution of one of them will only impact the evolution of a small group of other ones. A total of 910 projects were assigned to this cluster and a majority of *Plateau* patterns are assumed in the evolution of the metrics. Finally, the plot of *Cluster_2* is between the other two clusters. On one end, it shows a faster growth for some metrics than *Cluster_1*, while it also shows a more limited metrics correlation than *Cluster_0*. A total of 277 open-source repositories was assigned to this cluster and it can be expected to see a combination of *Shallow* and *Plateau* patterns in the metrics evolution. The discussed results and patterns assumptions are confirmed by the distributions shown in table 4.2.

Clusters / Patterns	Steep	Shallow	Plateau
Cluster_0	0.198	0.512	0.29
Cluster_1	0.088	0.047	0.865
Cluster_2	0.088	0.38	0.531

Table 4.2: Distribution of metrics evolution patterns over the repositories clusters

The discussed results provide insights into what the patterns modeling and the multivariate time series methodologies from the framework can achieve. The identified evolution patterns give researchers the means to interpret the evolutionary behavior of open-source projects, while the clustering allows the identification of similar ones to draw further comparisons.

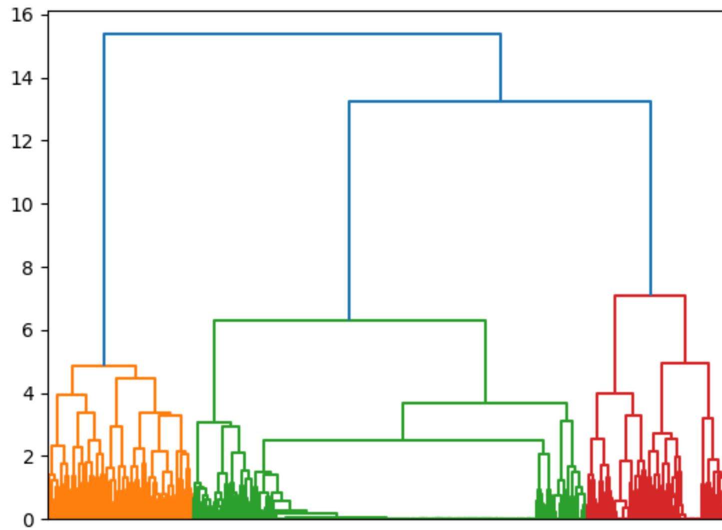

Figure 4.2: Dendrogram representing the hierarchical clustering of open-source software projects based on their metrics evolution patterns. Source: Thesis author

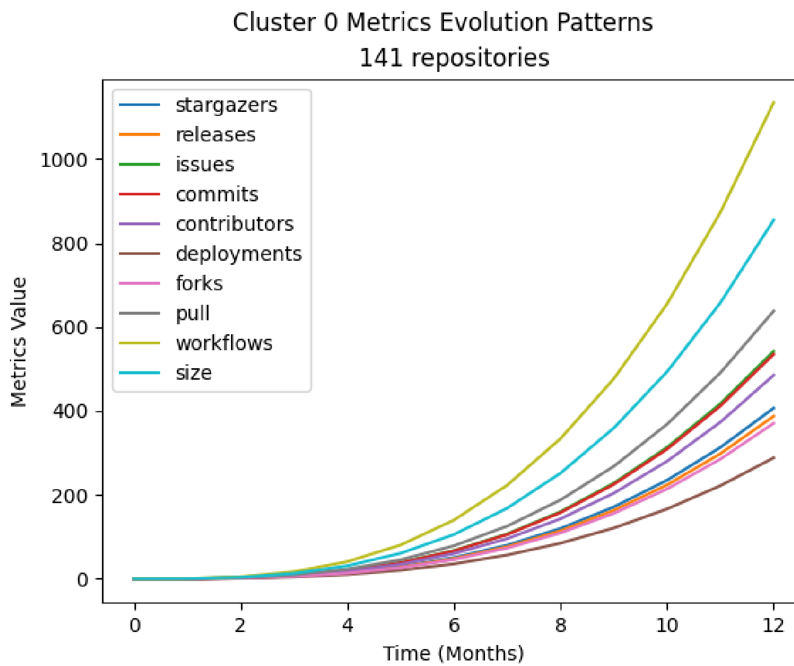

Figure 4.3: Average metrics evolution patterns for cluster 0. Source: Thesis author

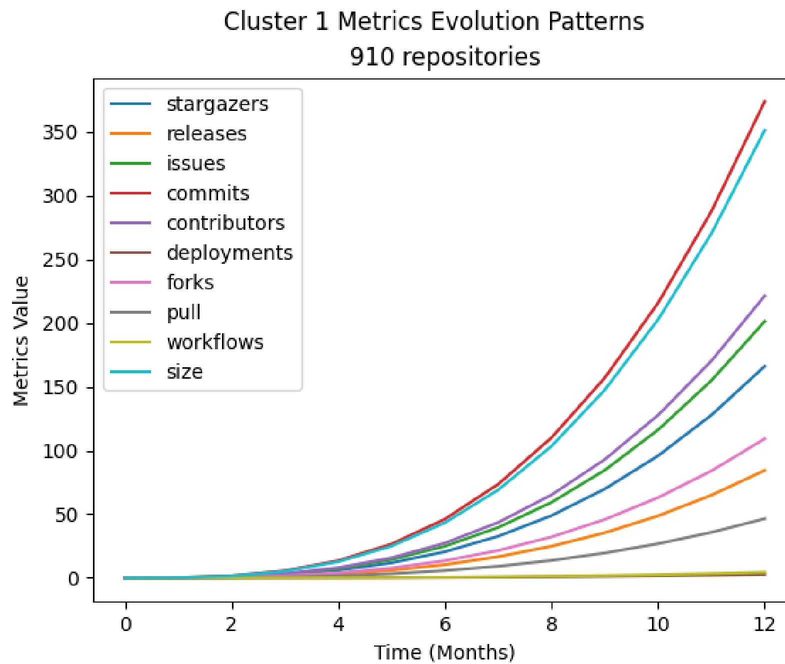

Figure 4.4: Average metrics evolution patterns for cluster 1. Source: Thesis author

4.1.2 RQ1.2: How Well Do the Forecasting Models Fit the Historical Data?

The evaluation of the multivariate forecasting models from chapter 3.3.4 is performed by training a new model for all metrics of 10 randomly picked open-source projects from the dataset using an increasing amount of historical data. This amount of projects was chosen as it suited the computational resources available for the experiments. The forecast metrics values are then segmented and mapped to a series of patterns. The obtained patterns are then compared to the ground truth, which is the sequence of patterns derived from the originally collected time series data. The quality of predictions and how well the models fit the data are evaluated using the following scores:

- **Performance**: The performance is the ratio between the matching monthly pattern predictions and the total monthly patterns. This aims to assess the ability of the forecasting models to predict values that can be mapped to the same patterns as the original time series.
- **Precision**: The precision is the fraction of the forecast relevant break points (detected within 12 months from the true values) and their total amount. This aims to assess how many detected break points from the forecast time series are relevant compared to the ones from the historical data [94].
- **Recall**: The recall is the fraction of the forecast relevant break points (detected within 12 months from the true values) and the total amount of break points detected in the

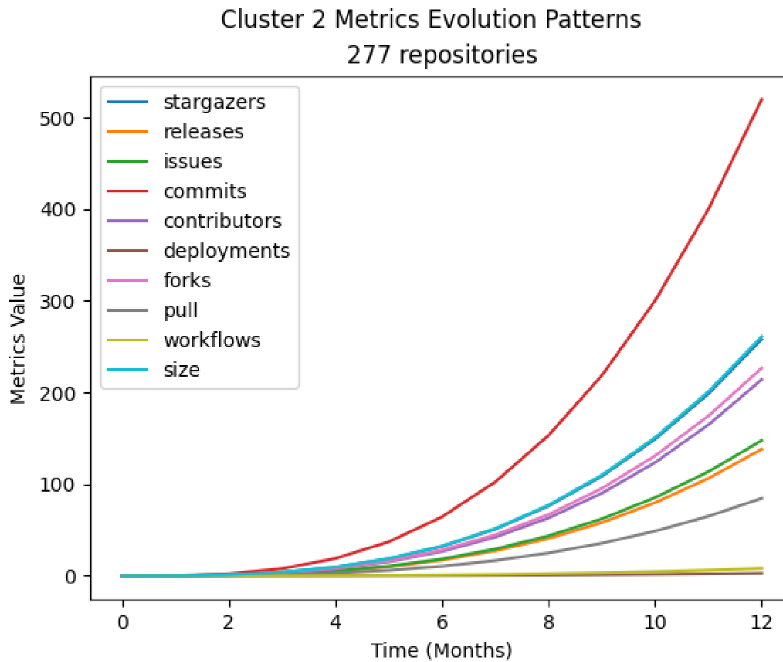

Figure 4.5: Average metrics evolution patterns for cluster 2. Source: Thesis author

original data. The aim is to assess how many relevant break points have been detected from the forecast time series [94].

- **Random Index (RI)**: The random index measures the similarity between two time series segmentation by comparing the detected break points. A value of 0 represents a total disagreement, while 1 represents a total agreement. The aim is to assess the similarity of the break points derived from the forecast time series and the ones from the historical data [25].

- **Deviation**: The deviation measures the average difference between the monthly predictions and the original data. The patterns defined in research question 1.1 4.1.1 are mapped as follows: *Steep* as 0; *Shallow* as 1; *Plateau* as 2. This aims to assess how far the patterns of the predicted time series are from the ones of the historical data.

- **Hausdroff Score**: The Hausdroff score computes the worst prediction error when comparing two time series segmentation. This aims to assess the maximum misalignment (in months) that occurs between the break points derived from the forecast time series and the ones from the historical data [71].

- R^2 **score**: The coefficient of determination provides a measure of how well the historical time series data are replicated by the forecasting model, based on the proportion of total variation of outcomes provided by the model [30].

Months	Performance	Precision	Recall	RI	Deviation	Hausdorff	R^2
1	0.777	0.474	0.549	0.774	0.238	21.85	0.553
2	0.782	0.469	0.576	0.774	0.229	22.35	0.555
4	0.802	0.469	0.533	0.775	0.215	21.95	0.559
6	0.759	0.485	0.586	0.774	0.257	21.75	0.559
12	0.776	0.504	0.578	0.776	0.224	21.65	0.657
24	0.792	0.504	0.577	0.776	0.224	21.65	0.657
48	0.771	0.506	0.589	0.797	0.239	19.6	0.692
72	**0.861**	**0.562**	**0.613**	**0.844**	**0.149**	**16**	**0.739**

Table 4.3: Average metrics patterns prediction accuracy measures and model R^2 score evaluated on 10 randomly picked open-source projects

The results of the evaluation procedure are shown in table 4.3, where the average scores of the 10 randomly picked projects are reported for each of the amount of training history data used. In terms of models fitting, the R^2 scores indicate that more than 12 months of history start providing a better match between the forecast data and the historical data, with the value steadily increasing as more known project history is included in the training data. Using more historical data also brings the patterns predicted from the forecast time series more in line as shown by the increasing performance and decreasing deviation scores. In this case, it is worth pointing out how even the smallest amount of historical data (1 month) shows about 77% of correct pattern prediction and a small value of pattern deviation. This might suggest that, if the only goal is the pattern prediction, even a contained size of historical data might suffice.

In terms of the time series segmentation and the comparison of the break points detected from the forecast time series and the historical data, the random index score shows a good overall similarity even for small amounts of training history. As more training data is used, the score shows better values. On the other hand, the precision and recall do not increase as much as other scores and just reach around 60% in the best case when 72 months of historical data is used for the training. This might be related to the fact that the average Hausdorff error, despite decreasing, is still at 16 months in the best case. This value is higher than the margin of 12 months allowed for the precision and recall scores and probably affects their outcomes.

Overall, the evaluation output shows that the multivariate forecasting models provide good results when it comes to predicting the evolution patterns, which can be done with limited amounts of training data. For this task, the inverse relationship between the patterns prediction performance and deviation is directly linked to how well the model fits the data as shown in figure 4.6. On the other hand, more training data is needed to obtain more accurate value predictions and better break points detection.

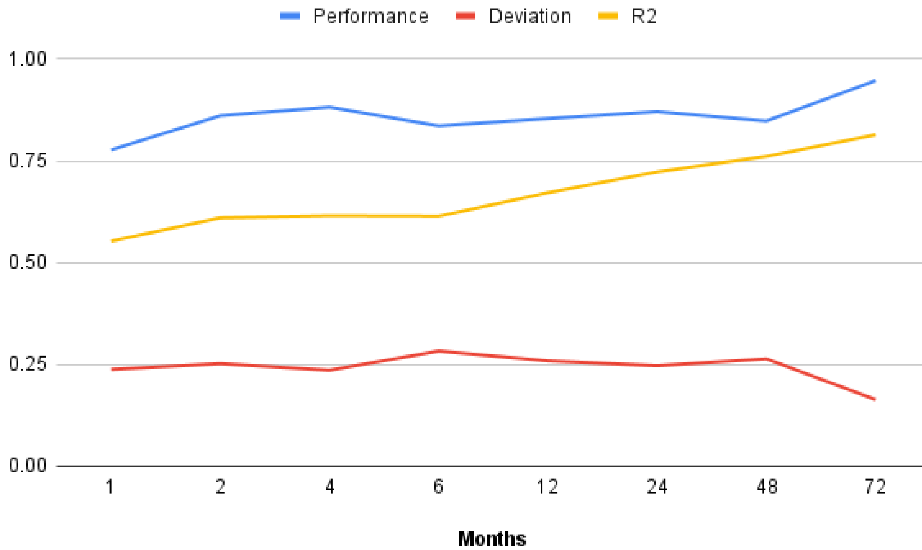

Figure 4.6: Average patterns prediction performance, deviation and model R^2 scores over training months size evaluated on 10 randomly picked open-source projects. Source: Thesis author

4.1.3 RQ1.3: How do the Metrics Influence Each Other in Project Evolution?

From the trained forecasting models described in chapter 3.3.4, it is possible to extract the relevance of each feature in the prediction task. This allows the inspection of which metric influences the most the growth of another. This research question aims to provide an overview of the features importance for each metric forecasting model and how the removal of features affects the accuracy of such models.

The features importance is obtained from the XGBoost regression models trained following the methodology from chapter 3.3.4 using their native *features_importances_* function[1]. The measured importance corresponds to the *Gain*, which represents the average improvement in loss brought by a feature. In other words, it tells how much a feature helps to make accurate predictions using the training data [91]. To evaluate the effect of the features importance in the metrics forecasting models, an ablation study is conducted by performing the same tasks described in chapter 4.1.2 for research question 1.2 using the best results of $N = 72$ as baseline and the same 10 random projects sample. The tasks are repeated for a smaller amount of the most important features for each model in each iteration, namely 9 (baseline), 6, 3, and 1. This ablation study aims to assess the impact of feature selection on the accuracy metrics described earlier.

[1] https://xgboost.readthedocs.io/en/stable/python/python_api.html#xgboost.dask.DaskXGBRegressor.feature_importances_

Models / Features	**Stargazers**	**Issues**	**Commits**	**Contributors**	**Deployments**	**Forks**	**Pull Requests**	**Workflows**	**Releases**	**Size**
Stargazers	-	0.11	**0.2**	0.13	0.01	0.13	0.08	0.13	0.08	0.13
Issues	0.09	-	0.1	0.11	0.01	0.09	**0.3**	0.07	0.09	0.1
Commits	0.07	0.1	-	0.12	0.02	0.08	0.06	0.14	0.1	**0.3**
Contributors	0.08	0.08	0.24	-	0.02	0.08	0.09	0.05	0.09	**0.27**
Deployments	0.19	**0.2**	0.06	0.04	-	0.03	0.01	0.0001	0.04	0.1
Forks	0.12	0.13	0.12	0.11	0.045	-	**0.16**	0.1	0.09	0.14
Pull Requests	0.11	**0.15**	0.11	0.11	0.07	**0.15**	-	0.13	0.08	0.09
Workflows	0.11	0.12	0.12	0.11	0.03	0.06	**0.37**	-	0.04	0.04
Releases	0.13	0.13	0.14	0.11	0.06	0.09	0.04	**0.18**	-	0.12
Size	0.06	0.09	**0.37**	0.12	0.04	0.11	0.07	0.06	0.08	-
Average Importance	0.1	**0.11**	**0.15**	0.1	0.03	0.08	**0.12**	0.09	0.07	**0.13**
Most Important	0	2	2	0	0	1	**3**	1	0	2

Table 4.4: Average features importance for the metrics forecasting models including how many times they are the most important

From the values shown in table 4.4, it can be seen to which extent each feature plays a role in the prediction accuracy of other ones. From these findings, insights on how metrics influence each other in the evolution of open-source software can be drawn. For example, the number of pull requests is what affects the forecast of the number of issues the most, which can be translated to actual dynamics of open-source project development as more of the former is usually linked to a fix of the latter. Additionally, from the average importance, it emerges that the number of commits, pull requests, issues and the project size are the four features that yield the highest gain in the metrics forecasting models. The number of pull requests also appears to be the most frequently important feature across the models. These results can also be used to revise the metrics selection process as the ones with lower importance might be omitted from the evolution analysis.

Table 4.5 shows the results of the ablation study compared to the baseline results when all 9 features are used. In general, a reduction of features seems to improve specific metrics depending on the extent of the omissions. When only 6 features are used, the patterns prediction scores show slight improvements that can be related to the fact that the features that yield the smallest gain are not accounted for anymore. A more drastic reduction does not seem to improve these metrics but it benefits the models' ability to fit the training data,

Features	Months	Performance	Precision	Recall	Random Idx	Deviation	Hausdorff	R^2
9 (Baseline)	72	0.861	0.562	0.613	0.844	0.149	16	0.739
6	72	0.877	**0.585**	0.613	**0.849**	**0.13**	16.25	0.74
3	72	0.874	0.566	**0.623**	0.844	0.14	**14.8**	0.739
1	72	**0.878**	0.553	0.615	0.842	0.136	15.55	**0.85**

Table 4.5: Average metrics patterns prediction accuracy measures and model R^2 score evaluated on 10 randomly picked open-source projects using a decreasing amount of features. Features=9 and Months=72 are used as baseline

as seen on the R^2 score.

Overall, the assessment of the features importance highlighted how it is possible to inspect how metrics influence each other through the evolution of an open-source project. From this, researchers can derive the relationships between metrics and inform their selection. On top of this, the ablation study showed how a feature selection based on their importance might benefit some tasks in the study of open-source software evolution, especially when it comes to use the forecasting models to predict growth patterns.

4.2 RQ2: How Much History is Needed for the Patterns Predictions?

This research question aims to take a deeper look at the prediction of monthly evolution patterns to understand how many months of historical patterns have to be known to perform accurate forecasting. The evaluation is based on the comparison of the following pattern prediction models using an increasing amount of known monthly patterns:

- **Baseline**: The model used as a baseline simply repeats the latest known pattern until the last month.
- **N-grams**: The N-grams model performs an N-gram analysis of the patterns sequences to calculate the probabilities with which one pattern appears [4][90]. The next pattern is chosen by calculating the sequence probability by applying the bi-grams probabilities from table 4.6 to the chain rule formula[90]:

$$P(w_{1:n}) = P(w_1)P(w_2|w_1)P(w_3|w_{1:2})...P(w_n|w_{1:n}) = \prod_{k=1}^{n} P(w_k|w_{1:k-1})$$

The input data for the models is the metrics patterns sequences from the collected open-source projects. The data is split into training and test sets using an 80%-20% ratio. The training set is used to build the bi-grams probabilities for the N-grams models, which are shown in table 4.6. Due to its simplicity, the Baseline model is directly applied to the test set as no training is needed for it. Both models are then evaluated by predicting future patterns

Pattern	Plateau	Shallow	EOS	Steep	Plateau	Shallow	EOS	Steep
Plateau	**0.983**	0.003	0.012	0.001	**190183**	659	2370	276
Shallow	0.045	**0.984**	0.009	0.002	632	**138720**	1322	327
EOS	0.453	0.276	-	0.271	1880	1144	-	1122
Steep	0.046	0.028	0.026	**0.9**	793	478	454	**15528**

Table 4.6: Bi-grams probabilities and frequencies with EOS indicating one of the ends of a sequence (first or last)

given an increasing amount of known previous ones. The original pattern sequences are used to evaluate the quality of the forecasts, which is expressed using the following metrics:

- **Performance**: The performance is the ratio between the matching monthly pattern predictions and the total monthly patterns. This aims to assess the ability of the forecasting models to predict values that can be mapped to the same patterns as the original time series.
- **Deviation**: The deviation measures the average difference between the monthly predictions and the original data. The patterns defined in research question 1.1 4.1.1 are mapped as follows: *Steep* as 0; *Shallow* as 1; *Plateau* as 2. This aims to assess how far the patterns of the predicted time series are from the ones of the historical data.

Months	N	Performance N-grams	Performance Baseline	Deviation N-grams	Deviation Baseline
1	2	0.849	0.848	0.222	0.223
2	3	0.85	0.848	0.221	0.223
3	4	0.851	0.848	0.219	0.223
6	7	0.855	0.848	0.214	0.223
12	13	0.931	0.924	0.083	0.091
24	25	**0.945**	**0.938**	**0.071**	**0.076**

Table 4.7: Patterns prediction performance and deviation over increasing N comparison between N-grams and baseline models

The evaluation results are shown in table 4.7 and figure 4.7. From the numbers and the plot, it can be seen how the predictions get more accurate as more historical patterns are known with a significant increase after $N = 13$. This also coincides with a more than half reduction of the deviation value. In terms of the comparison between the two models, the results suggest that there is not much difference between the Baseline and the N-grams. This can be caused by the nature of the input data itself since table 4.6 shows that a pattern is very likely followed by one of the same type. Therefore, even repeating the latest known pattern or having little historical data yields good prediction accuracy. In light of this, when

the dataset presents sequences of frequent pattern repetitions, using a simple model with a limited amount of historical patterns can be enough to infer future patterns.

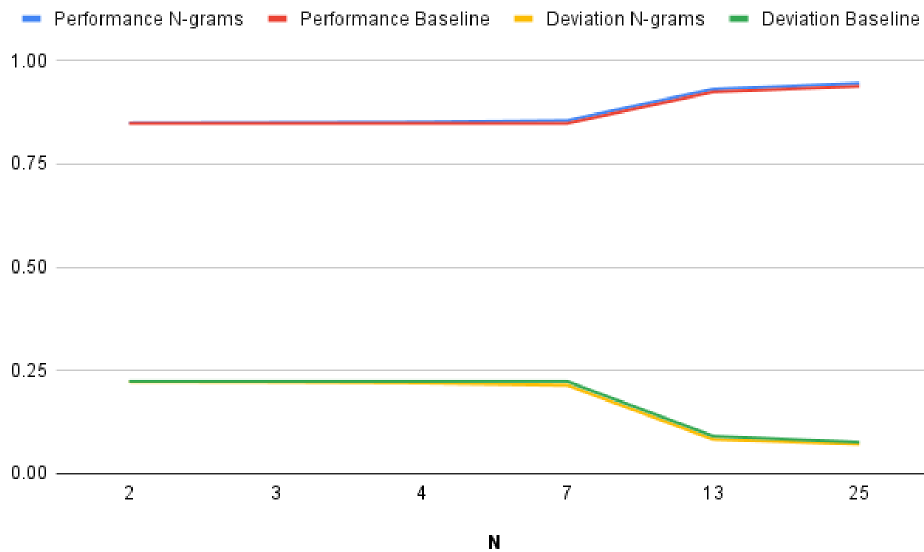

Figure 4.7: Patterns prediction performance and deviation over increasing N. Source: Thesis author

Although the results obtained in the previous iteration of the models evaluation are good and provide useful insights, it is important to take a step further in answering this research question. As mentioned before, table 4.6 shows how most of the patterns sequences are made of the same types of patterns. These probabilities suggest that the models that were previously evaluated are good when it comes to predicting the same pattern over and over. However, the changes in pattern type are the ones that reflect useful information about the evolution of an open-source project. Therefore, it is worth inspecting how good the two models are in predicting future patterns for sequences that are not just made of one pattern type. This is achieved by removing from the test set the patterns sequences that present no changes in pattern types. This validation set is then used to evaluate the accuracy of the two models.

The results in table 4.8 and figure 4.8 show how the models perform when it comes to predicting sequences of patterns that change type at some point in time. The first thing to notice is how the performance and the deviation are worse than the previous results, especially for lower amounts of known patterns. A significant improvement is present when more than 12 months of history are known both in terms of performance and deviation. The trend of having better results as N increases is then confirmed here as well. Additionally, the results between the two models are not very far apart. Although the gap is slightly wider compared to the previous iteration, the simpler model might still be enough to obtain accurate pattern predictions. The main difference with the former experiment is that, in this

Months	N	Performance N-grams	Performance Baseline	Deviation N-grams	Deviation Baseline
1	2	0.648	0.646	0.518	0.522
2	3	0.651	0.646	0.515	0.522
3	4	0.654	0.646	0.511	0.522
6	7	0.662	0.646	0.499	0.522
12	13	0.841	0.823	0.192	0.213
24	25	**0.874**	**0.855**	**0.163**	**0.178**

Table 4.8: Patterns prediction performance and deviation over increasing N comparison between N-grams and baseline models considering pattern changes only

case, a higher amount of known patterns is required for good results.

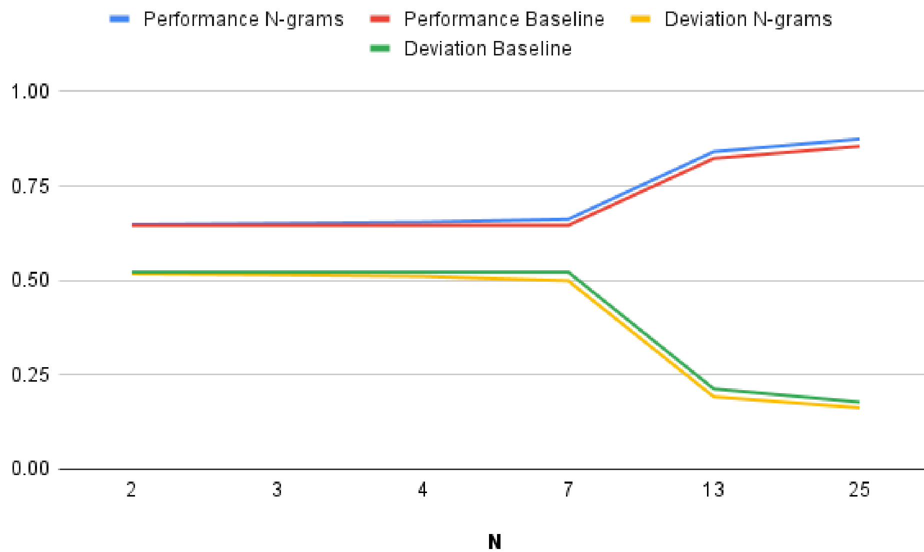

Figure 4.8: Patterns prediction performance and deviation over increasing N considering only patterns changes. Source: Thesis author

Overall, the performed evaluation suggests that the amount of history needed to predict patterns is dependent on the goal of the forecasting, given the fact that repetitions of the same pattern types occur more often. When it comes to just predicting the patterns sequence without focusing on type shifts, a small amount of known patterns ($N = 2$) and a simple model (Baseline) is enough. On the other hand, when the focus is to accurately predict changes in patterns, the simple model (Baseline) still suffices but the more previous patterns are known ($N >= 13$) the better it is.

4.3 RQ3: How Reliable is the Break Points Detection?

This research question aims to assess if the detected break points from the time series segmentation (chapter 3.3.1) can be traced back to relevant real world events that occurred throughout the history of an open-source project. The inspection of the break points and real events alignment is firstly carried out in section 4.3.1 on 10 randomly picked projects. For the projects with little to no events found, a further inspection is performed by introducing additional metrics in section 4.3.2.

4.3.1 RQ3.1: How Well Do the Detected Break Points Align With Real Events?

The evaluation of the break points alignment with real events is performed on a random sample of 10 of the most active open-source projects from the collected dataset. This amount of projects was chosen as it suited the computational resources available for the experiments. The break points generated from the time series segmentation process, which serve as time boundaries for the metrics growth patterns, are validated by confirming relevant changes in the projects' GitHub activities around the indicated date. Additionally, where possible, events that reflect the break points beyond GitHub are reported. The break points accuracy is also validated the other way around by inspecting if relevant activities on a project outside of GitHub (e.g. release announcement on website, external bug tracker, issues blog) also correspond to a break point around the same time. Overall, 5 projects are assessed using the former approach (PatternFly-React, Conan, CockroachDB, pip, Woocommerce), while the other 5 are assessed using the latter (NextCloud, Nixpkgs, Gutenberg, Salt, AWX).

For this evaluation, the focus is directed to the following metrics: *releases*; *issues*; *pull_requests*; *workflows*. This is because the manual analysis of such metrics in the GitHub user interface (UI) is straightforward, unlike the other ones (e.g. *commits* and *contributors* only show statistics for the last 14 days, *deployments* are not searchable by date). For each analyzed project, the plot of the merged metrics curve is shown alongside a table that summarizes the metrics patterns and break points. The patterns in the tables are color-coded as follows to improve readability: *Steep* (Red); *Shallow* (Orange); *Plateau* (Yellow).

patternfly/patternfly-react PatternFly-React provides a React component to PatternFly, an open-source design system that enables the creation of enterprise products. Started in early 2017, the project gathered more than 700 stargazers, 180 contributors and was forked more than 300 times. At the time of writing, more than 150 issues are still open (more than 4500 closed overall) together with more than 20 pull requests (almost 5300 closed overall)[2].

The merged curve in figure 4.9 shows how the development activities for the project followed a steady growth until more recent years when a sharp increase in the pace was detected. This is reflected by the change of pattern from *Shallow* to *Steep* in metrics like the number of open *issues*, *pull_requests* and changes *size* after July 2022. By inspecting the number of opened *issues* around the July 2022 break point from project's GitHub issues

[2] `https://github.com/patternfly/patternfly-react`

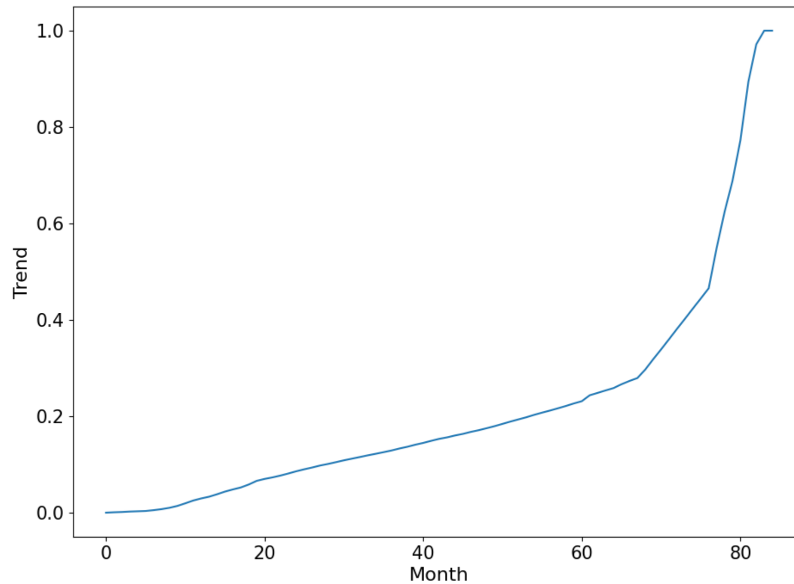

Figure 4.9: Merged metrics curve for the patternfly/patternfly-react project. Source: Thesis author

page, it can be seen how they started accumulating after the beginning of 2020[3]. This can be interpreted as the beginning of the growth pattern shift for this metric and it can be related to the need of addressing the technical debt that piled up over the years as a third of the opened *issues* before July 2022 are marked for that scope. Additionally, the *Steep* patterns that follow can be attributed to a rise in the amount of breaking changes introduced and detected bugs, which concern more than a half of the newly opened *issues* until March 2024[4]. This is in line with the changelogs of the releases created from March 2024 until the time of writing, where less than 10 of the more than 50 releases are not about bug fixes[5].

The shift in pattern from *Shallow* to *Steep* after May 2023 for the opened *pull_requests* is linked to a series of dependency updates[6]. The number of *releases* growth registered a break point around February 2022. As the project holds more than 10000 releases on GitHub, this shift in the growth rate is confirmed by the fact that about 8000 releases were created in the

[3] `https://github.com/patternfly/patternfly-react/issues?q=is%3Aissue+is%3Aopen+created%3A%3C2022-07-01`

[4] `https://github.com/patternfly/patternfly-react/issues?q=is%3Aissue+is%3Aopen+created%3A2022-07-01..2024-03-01`

[5] `https://github.com/patternfly/patternfly/releases?q=created%3A2024-03-01..2024-06-01&expanded=true`

[6] `https://github.com/patternfly/patternfly-react/pulls?page=1&q=is%3Apr+is%3Aopen+created%3A%3E%3D2023-05-01`

	stargazers	releases	size	commits	contributors	deployments	issues	forks	pull_requests	workflows
03/2017										
05/2018								0.18		
10/2018	0.21				0.12					
03/2019				0.22				0.36		
04/2021	0.64							0.63		
09/2021					0.57					
02/2022		0.53		0.77						
07/2022					0.74		0.21	0.84		
05/2023			0.67				0.94		0.06	0
03/2024	1	1	1	1	1	0	1	1	1	1

Table 4.9: Metrics patterns for the patternfly/patternfly-react project

first 5 years[7], while more than 6500 were added in the last 2 years[8]. Although the pattern changes have been successfully linked to GitHub, no external events were detected for this project given its break points.

conan-io/conan Conan is a decentralized package manager that allows developers to publish and install C and C++ binaries. Started in late 2015, the project gathered more than 7500 stargazers, 350 contributors and was forked more than 900 times. At the time of writing, more than 1500 issues are still open (more than 8500 closed overall) together with more than 40 pull requests (almost 5900 closed overall)[9].

The merge curve from figure 4.10 shows that the overall growth of the project has been consistent throughout its lifetime with a sharper rise in recent times. This is reflected in table 4.10, where the majority of metrics follow a series of *Shallow* patterns before changing into *Steep* ones in the last segment. The shift in pattern for the number of opened *issues* is detected around July 2022. Before this time, the growing amount of open *issues* was evenly split between support questions, features request and bugs[10]. After this, a significant increase in the number of support questions drove the growth pattern of these metrics to *Steep*. Although the number of bugs and feature requests also increased, the support questions accounted for about half of the newly opened *issues* after July 2022. Still, the

[7] `https://github.com/patternfly/patternfly-react/releases?q=created%3A%3C2022-02-01&expanded=true`

[8] `https://github.com/patternfly/patternfly-react/releases?q=created%3A%3E%3D2022-02-01&expanded=true`

[9] `https://github.com/conan-io/conan`

[10] `https://github.com/conan-io/conan/issues?q=is%3Aissue+is%3Aopen+created%3A%3C2022-07-01`

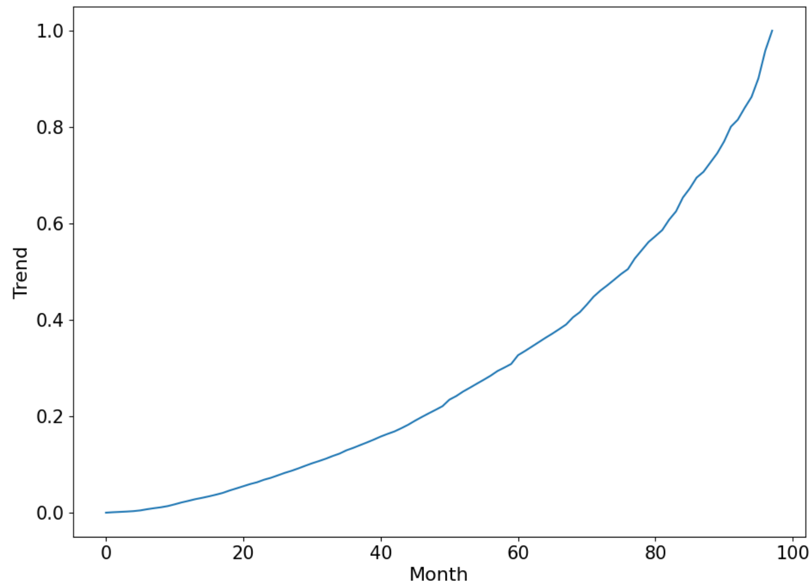

Figure 4.10: Merged metrics curve for the conan-io/conan project. Source: Thesis author

increase in the number of features requests and bugs reports contributed to the faster growth of the opened *pull_requests* after May 2023[11]. These patterns are also reflected in the features announcements of the following months: agnostic deployment of dependencies in May 2023[12]; bulk uploads and downloads in June 2023[13]; CLion Conan plugin in August 2023[14].

The break points for the number of *releases* can be inspected by looking at how many of them occurred before and after the marked dates. Following the almost 40 releases between December 2015 and December 2017[15], about 60 releases were added until March 2019[16], which reflects an increase in the release per year frequency. Following March 2019, the growth rate slows down as about 180 releases are added until January 2024. Although all the segments are classified as *Shallow*, it is still possible to verify the reliability of the detected break points for this metric. Overall, the break points inspection for this project led to successful findings of real events to link to pattern changes.

[11]https://github.com/conan-io/conan/pulls?q=is%3Apr+is%3Aopen+created%3A%3E%3D2023-05-01+

[12]https://blog.conan.io/2023/05/23/Conan-agnostic-deploy-dependencies.html

[13]https://blog.conan.io/2023/06/28/Conan-bulk-package-operations.html

[14]https://blog.conan.io/introducing-new-conan-clion-plugin/

[15]https://github.com/conan-io/conan/releases?q=created%3A%3C2017-12-01&expanded=true

[16]https://github.com/conan-io/conan/releases?q=created%3A2017-12-01..2019-03-01&expanded=true

	stargazers	releases	size	commits	contributors	deployments	issues	forks	pull_requests	workflows
12/2015										
07/2017								0.12		
12/2017		0.21								
10/2018				0.37						
03/2019		0.42								
08/2019					0.38					
01/2020	0.44							0.43		
06/2020					0.52		0.17			
11/2020									0.06	
09/2021								0.65		
02/2022				0.77						
07/2022							0.57			
12/2022	0.86									
05/2023			0.86		0.89		0.8	0.9	0.47	
01/2024	1	1	1	1	1	0	1	1	1	0

Table 4.10: Metrics patterns for the conan-io/conan project

cockroachdb/cockroach CockroachDB is a cloud-based distributed SQL database designed to build and manage scalable data-intensive applications. Started in early 2014, the project gathered more than 29000 stargazers, 700 contributors and was forked more than 3600 times. At the time of writing, more than 5000 issues are still open (more than 57000 closed overall) together with more than 900 pull requests (almost 61000 closed overall)[17].

The steady growth of the project's metrics is shown in figure 4.11 with a spike detected at the very end. This is also confirmed in table 4.11, where the last segment of most metrics is classified as *Steep* growth. The number of *issues* opened before March 2023 are mostly linked to reported bugs and new features[18], which cover about half and a quarter of them respectively. The change from *Shallow* to *Steep* in the latest segment is because more than half of the overall opened *issues* are clustered after March 2023. These new *issues* are mostly characterized by bugs and enhancement related to the code-base[19]. A similar pattern is seen in the number of opened *pull_requests*, where more than a half of them were created

[17] https://github.com/cockroachdb/cockroach

[18] https://github.com/cockroachdb/cockroach/issues?q=is%3Aissue+is%3Aopen+created%3A%3C2023-03-01++

[19] https://github.com/cockroachdb/cockroach/issues?q=is%3Aissue+is%3Aopen+created%3A%3E%3D2023-03-01

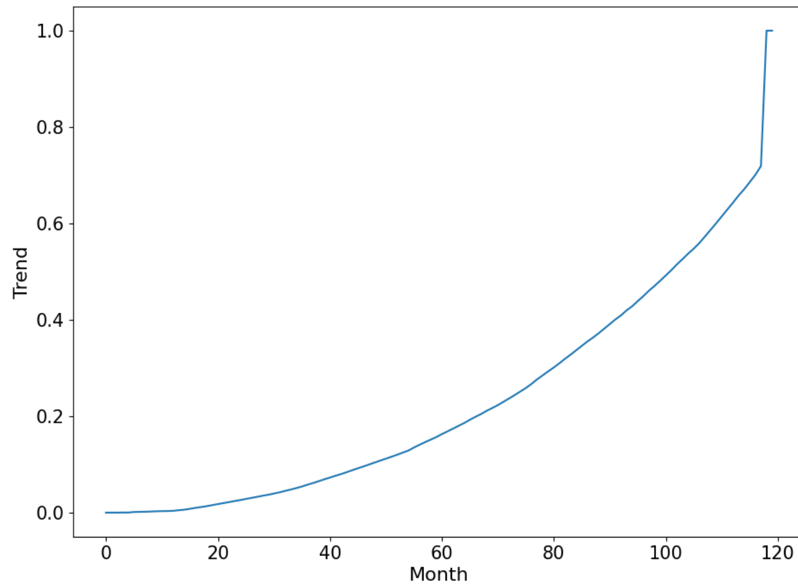

Figure 4.11: Merged metrics curve for the cockroachdb/cockroach project. Source: Thesis author

after March 2023[20,21]. In this case, the vast majority of the *pull_requests* are linked to pending releases, which can also be attributed to the increasing number of *issues* that were being addressed. An example of a set of features that might have contributed to these shifts in patterns is the release of a multi-region deployment feature in May 2023[22], which likely required a considerable amount of development activities and is followed by a series of reports for bugs and enhancements. Although the pattern changes have been successfully linked to GitHub, only 1 external event was detected for this project given its break points.

pypa/pip pip is the package installer for Python, which allows the packages retrieval from the Python Package Index and other indexes. Started in early 2011, the project gathered more than 9000 stargazers, 680 contributors and was forked more than 3000 times. At the time of writing, more than 900 issues are still open (more than 6200 closed overall) together with more than 100 pull requests (almost 5400 closed overall)[23].

From the merged curve in figure 4.12, it can be seen how the metrics growth started at a

[20] https://github.com/cockroachdb/cockroach/pulls?q=is%3Apr+is%3Aopen+created%3A%3C2023-03-01++
[21] https://github.com/cockroachdb/cockroach/pulls?q=is%3Apr+is%3Aopen+created%3A%3E%3D2023-03-01+
[22] https://www.cockroachlabs.com/blog/cockroachdb-23-1-release/
[23] https://github.com/pypa/pip

	stargazers	releases	size	commits	contributors	deployments	issues	forks	pull_requests	workflows
02/2014										
04/2015	0.15									
05/2017	0.35							0.24		
10/2017				0.35						
08/2018							0.03			
01/2019	0.52							0.41		
09/2020									0.15	
02/2021					0.49			0.68		
12/2021	0.82									
03/2023			0.87		0.73		0.69		0.6	
01/2024	1	0	1	1	1	0	1	1	1	0

Table 4.11: Metrics patterns for the cockroachdb/cockroach project

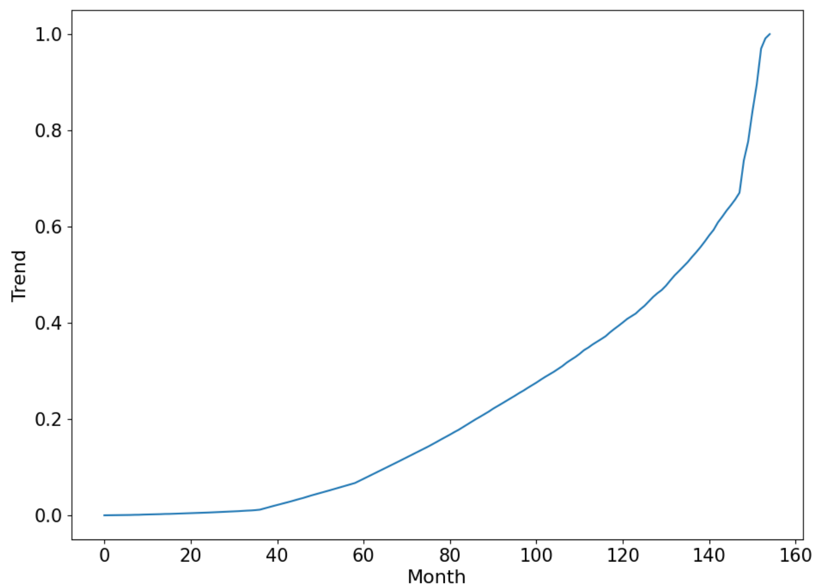

Figure 4.12: Merged metrics curve for the pypa/pip project. Source: Thesis author

slow pace before picking up and reaching a sharp rise in recent months. Table 4.12 reflects this behavior as many metrics follow long *Shallow* patterns before switching to a *Steep*

pattern in the last segments. The number of opened *issues* before March 2023 is driven by an even split of bug report and feature requests[24,25,26]. The split continues until January 2024, with the main difference that more bugs are reported rather than feature requests[27]. This last segment is marked as *Steep* since *issues* are opened at a faster rate than the previous ones. The same behavior can be found in the growth of the opened *pull_requests* before and after March 2023[28,29], where the split between bug fixes and features implementation reflects the findings discussed for the *issues*. Finally, the introduction of CI/CD pipelines shown in the *Steep* growth of the number of *workflows* is visible by the more than 7000 GitHub actions run after March 2023[30]. Although the pattern changes have been successfully linked to GitHub, no external events were detected for this project given its break points.

	stargazers	releases	size	commits	contributors	deployments	issues	forks	pull_requests	workflows
03/2011										
11/2014				0.19						
03/2018	0.43									
08/2018				0.43	0.49					
06/2019					0.56					
11/2019	0.61			0.61						
04/2020							0.38			
02/2021										
07/2021					0.78					
12/2021							0.65	0.68		
03/2023			0.9			0.55	0.88		0.69	0
01/2024	1	0	1	1	1	1	1	1	1	1

Table 4.12: Metrics patterns for the pypa/pip project

woocommerce/woocommerce Woocommerce is a customizable e-commerce platform built on WordPress that supports the creation of dedicated commerce solutions. Started

[24] https://github.com/pypa/pip/issues?q=is%3Aissue+is%3Aopen+created%3A%3C2020-04-01+ +

[25] https://github.com/pypa/pip/issues?q=is%3Aissue+is%3Aopen+created%3A2020-04-01. .2021-12-01

[26] https://github.com/pypa/pip/issues?q=is%3Aissue+is%3Aopen+created%3A2021-12-01. .2023-03-01+

[27] https://github.com/pypa/pip/issues?q=is%3Aissue+is%3Aopen+created%3A%3E%3D 2023-03-01+

[28] https://github.com/pypa/pip/pulls?q=is%3Apr+is%3Aopen+created%3A%3C2023-03-01+

[29] https://github.com/pypa/pip/pulls?q=is%3Apr+is%3Aopen+created%3A%3E%3D2023-03-01+

[30] https://github.com/pypa/pip/actions?query=created%3A%3E%3D2023-03-01+is%3Asuccess

in mid-2011, the project gathered more than 9100 stargazers, 1300 contributors and was forked more than 10700 times. At the time of writing, more than 3100 issues are still open (more than 24300 closed overall) together with more than 300 pull requests (almost 18800 closed overall)[31].

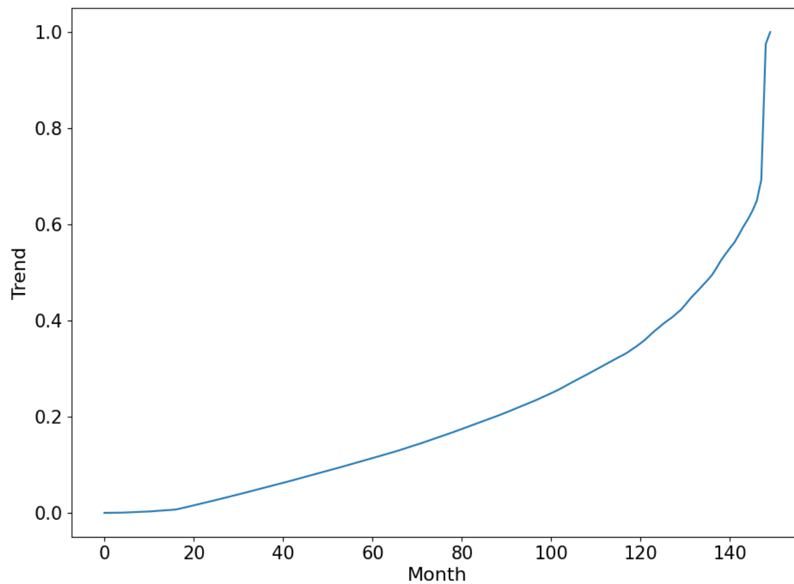

Figure 4.13: Merged metrics curve for the woocommerce/woocommerce project. Source: Thesis author

The merged curve in figure 4.13 shows how the metrics, after a slower start, grew steadily over time and picked up the pace in more recent months. Table 4.13 shows how the sequence of *Shallow* patterns with a *Steep* end for most metrics supports the curve behavior. The growth of the number of open *issues* before September 2020 is driven by the increasing reporting of bugs and requests for new features[32] which followed a significant increase until March 2023[33]. It is worth to report that, in July 2021, a SQL injection vulnerability was detected[34], which might have contributed to the metric growth. Since March 2023, the growth rate kept increasing by causing the switch to a *Steep* pattern until January 2024[35], still influenced by the reported bugs and features requests.

[31] https://github.com/woocommerce/woocommerce

[32] https://github.com/woocommerce/woocommerce/issues?q=is%3Aissue+is%3Aopen+created%3A%3C2020-09-01+

[33] https://github.com/woocommerce/woocommerce/issues?q=is%3Aissue+is%3Aopen+created%3A2020-09-01..2023-03-01

[34] https://blog.wpsec.com/woocommerce-unauthenticated-sql-injection-vulnerability-2/

[35] https://github.com/woocommerce/woocommerce/issues?q=is%3Aissue+is%3Aopen+created%3A%3E%3D2023-03-01+

	stargazers	**releases**	**size**	**commits**	**contributors**	**deployments**	**issues**	**forks**	**pull_requests**	**workflows**
08/2011										
04/2015								0.11		
10/2017				0.39	0.37					
08/2018								0.36		
01/2019				0.54						
06/2019					0.53					
09/2020	0.69						0.1	0.68		
12/2021		0.71								
03/2023			0.85	0.93	0.88		0.58		0.26	
01/2024	1	1	1	1	1	0	1	1	1	0

Table 4.13: Metrics patterns for the woocommerce/woocommerce project

In the matter of the opened *pull_requests* growth, it interesting to notice how the ones created before March 2023 are mostly labeled as contributions from the community[36]. As not all contributors are part of the core development team, requests coming from outside developers may be given lower priority and/or longer scrutiny. Following March 2023, the change into a *Steep* pattern is dictated by a fast rise in the number of bug fixes and features implementation[37]. The *Steep* patterns of both *issues* and *pull_requests* can also be linked to the announcements about the introduction of new features in the editor (November 2023)[38] and in the checkout experience (March 2023)[39]. Finally, the growth in the number of *releases* registered a break point around December 2021. Although all the patterns are classified as *Shallow*, it is possible to notice that the releases rate increased from almost 280 in about 10 years[40] to almost 200 in less than 3 years[41]. Overall, the break points inspection for this project led to successful findings of real events to link to pattern changes.

nextcloud/server NextCloud is an open-source alternative to commercial cloud storage services for both private individuals and enterprises. Started in mid-2016, the project gathered more than 25000 stargazers, 900 contributors and was forked more than 3000 times. At

[36] `https://github.com/woocommerce/woocommerce/pulls?q=is%3Apr+is%3Aopen+created%3A%3C2023-03-01+`

[37] `https://github.com/woocommerce/woocommerce/pulls?q=is%3Apr+is%3Aopen+created%3A%3E%3D2023-03-01++`

[38] `https://wptangerine.com/changes-in-woocommerce-editor/`

[39] `https://scottbolinger.com/headless-woocommerce-checkout/`

[40] `https://github.com/woocommerce/woocommerce/releases?q=created%3A%3C2021-12-01&expanded=true`

[41] `https://github.com/woocommerce/woocommerce/releases?q=created%3A%3E%3D2021-12-01&expanded=true`

the time of writing, more than 2300 issues are still open (more than 15700 closed overall) together with more than 450 pull requests (almost 25400 closed overall)[42].

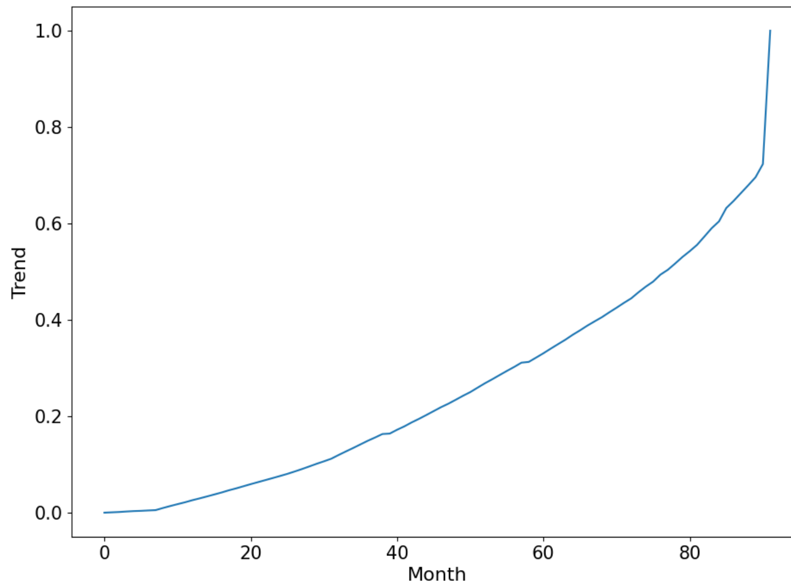

Figure 4.14: Merged metrics curve for the nextcloud/server project. Source: Thesis author

From the merged curve in figure 4.14, it can be seen how the growth pace increases after the first two years of the project's lifetime. The final part of the curve shows a steep increase, which is probably the result of stronger growth in several metrics. From table 4.14, it appears that the number of *releases*, *commits*, *issues*, *pull_requests*, *contributors* and changes *size* all follow a *Steep* pattern, which supports the behavior of the merged curve. Starting from the NextCloud help forum[43], three of the most active posts related to development activities are inspected to see if their timeline corresponds to any of the break points in table 4.14.

In May 2020, an announcement was made to ask users and contributors to help test the second candidate release for NextCloud version 19[44]. Until its closure in June 2020, the post gathered 79 responses from users and contributors who provided feedback, reported their open issues and proposed solutions. Therefore, it is expected that in this period between May 2020 and June 2020, an increase in *issues*, *pull_requests* and *commits* is registered. Additionally, further increases in these metrics alongside the number of *releases* can also appear as more feedback would come in after the new version is released. Table

[42]https://github.com/nextcloud/server
[43]https://help.nextcloud.com/
[44]https://help.nextcloud.com/t/nextcloud-19-rc2-is-here-help-us-test-it/81705

	stargazers	releases	size	commits	contributors	deployments	issues	forks	pull_requests	workflows
06/2016										
06/2018				0.3						
04/2019			0.22							
09/2019					0.37					
02/2020								0.42		
07/2020		0.11	0.4				0.21			
12/2020				0.6						
05/2021	0.55									
01/2023								0.86		
06/2023		0.83	0.91	0.91	0.75		0.75		0.58	
01/2024	1	1	1	1	1	0	1	1	1	0

Table 4.14: Metrics patterns for the nextcloud/server project

4.14 shows that, in the periods February-July 2020 and July-December 2020, the number of *releases*, *issues*, *commits* and changes *size* all register a shift in their growth pattern with the appearance of a break point. The next pattern for most of these metrics, despite having a stronger growth, is still classified as *Shallow* except for the change *size*, which switches from a *Plateau* pattern to a *Shallow* one.

Another help for testing post, which gathered 50 responses, was released in September 2023 about the first release candidate for NextCloud 27.1.2[45]. Here it can be assumed that a shift in the pattern of the development activities metrics should occur in the months before the announcement to reflect the work performed to prepare the release to test. From table 4.14, it can be seen how in the January-June 2023 period, the number of *releases*, *issues*, *commits*, *pull_requests* and changes *size* all have a break point that marks a change in the growth pace, which shifts to *Steep* for all of them. The stronger growth is also probably due to the activities introduced by users and contributors during the time the post was active (September-November 2023).

Much earlier in the development, another post was released in April 2018 to ask for help testing versions 12.0.7 and 13.0.2 of NextCloud[46]. The same metrics assumptions discussed before can be applied here for the time the post was active until August 2018. In table 4.14, the periods June 2016-June 2018 and June 2018-April 2019 show break points for the number of *commits* and changes *size*, which reflect the work on the code-base in the early releases. A shift in the growth pattern is not detected for other metrics at this time probably due to the lower amount of *contributors* involved (the post only gathered

[45] https://help.nextcloud.com/t/first-rc-of-nextcloud-27-1-2/171104
[46] https://help.nextcloud.com/t/help-testing-12-0-7-rc1-13-0-2-rc1/30585

20 replies). It is interesting to see how changes in the growth pace for other metrics only occur after the number of *contributors* growth rate changes itself. Overall, the break points inspection for this project led to successful findings of real events to link to pattern changes.

NixOS/nixpkgs Nixpkgs is a repository of software packages that can be installed with the Nix package manager on the NixOS Linux distribution. Started in mid-2012, the project gathered more than 15900 stargazers, 5000 contributors and was forked more than 12500 times. At the time of writing, more than 8000 issues are still open (more than 29600 closed overall) together with more than 5500 pull requests (almost 267100 closed overall)[47].

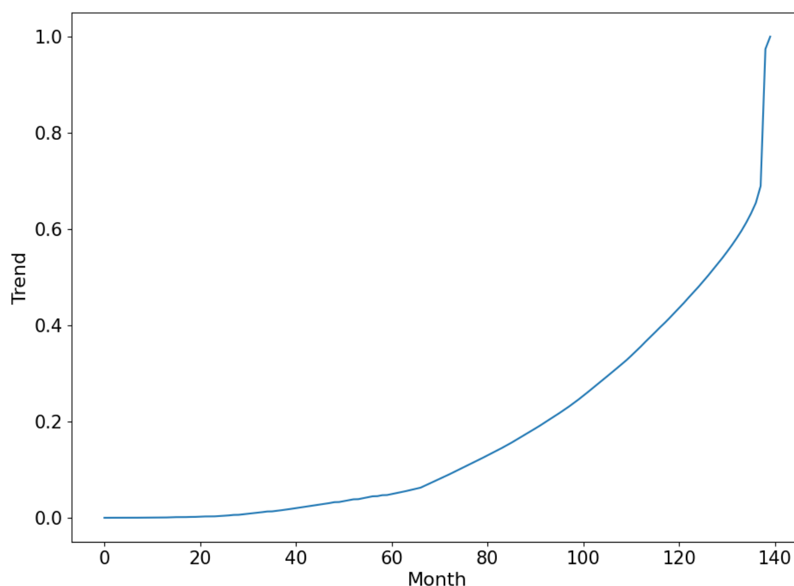

Figure 4.15: Merged metrics curve for the NixOS/nixpkgs project. Source: Thesis author

The merged curve in figure 4.15 shows how the project development started at a slow pace for the first few years before picking up significantly until recently. Table 4.15 shows that the number of *commits*, *issues*, *pull_requests*, *contributors*, *forks* and changes *size* all follow a *Steep* pattern in recent times, which is reflected in the behavior of the last part of the curve. Starting from the NixOS discussion forum[48], three of the most active posts related to development activities are inspected to see if their timeline corresponds to any of the break points in table 4.15.

A discussion about the possibility of the NixOS project moving out of GitHub after its acquisition from Microsoft was held between June 2018 and April 2020[49]. From this type

[47] https://github.com/NixOS/nixpkgs
[48] https://discourse.nixos.org/
[49] https://discourse.nixos.org/t/github-was-purchased-by-microsoft/313

	stargazers	**releases**	**size**	**commits**	**contributors**	**deployments**	**issues**	**forks**	**pull_requests**	**workflows**
06/2012										
02/2016			0.04							
12/2016								0.13		
03/2018				0.21						
01/2019	0.22									
06/2019					0.26					
09/2020					0.37					
12/2021	0.59				0.52					
03/2023	0.83		0.86	0.85	0.75		0.56	0.82	0.36	
01/2024	1	0	1	1	1	0	1	1	1	0

Table 4.15: Metrics patterns for the NixOS/nixpkgs project

of exchange, it is legitimate to expect an impact on the number of *contributors* involved in a project. Overall, since NixOS is still present and active on GitHub, it is safe to assume that no actions were taken to move it out and that the new ownership of GitHub did not completely discourage the community. However, it is interesting to see from table 4.15 how shifts in the growth of the number of *contributors* occurred in the periods of January-June 2019 and June 2019-September 2020, which overlap with the active time of the forum discussion. In particular, the metric goes from a *Shallow* pattern to a *Plateau* before picking up again to a *Shallow* growth from December 2021. This means that in a period of more than one year, very few new contributors joined the project. Although it is quite speculative to infer that the acquisition of GitHub by Microsoft drove this, it is an interesting coincidence that such discussion overlapped with flexion in the project's community expansion.

Another discussion was held between October 2020 and October 2021 about changing the releasing priorities of the project to provide more stable versions of the software[50]. As the exchange led to the decision to perform contributions to the main branch at a higher frequency, it is expected to see a change in the growth rate of several metrics in the periods after the discussion ended. From table 4.15, it can be seen how the number of *releases*, *commits*, *issues* and *pull_requests* all register a change in their growth pattern from March 2023. Although this is about one year and a half after the contributions style changed, it is important to remark that this took one year to discuss and would require time to be fully grasped by a community of over 5000 contributors.

The change of pattern of these metrics can also be attributed to another reason on top of the change in the contributions style. In February 2022, breaking changes were announced for the version of NixOS released at that time[51]. This meant that the pending fixes could

[50] https://discourse.nixos.org/t/what-should-stable-nixos-prioritize/9646
[51] https://discourse.nixos.org/t/breaking-changes-announcement-for-unstable/17574

have been affected and they needed to be revised. This discussion is still active as of May 2024. From this, it is expected that the development-related metrics would shift to faster growth as many changes need to be applied to existing open *issues* and related *pull_requests*. This can be seen in how the development metrics have been following a *Steep* growth in recent times (table 4.15). Overall, the break points inspection for this project led to successful findings of real events to link to pattern changes.

WordPress/gutenberg Gutenberg is an editor for WordPress that introduces a modular approach to building webpages. Started in early 2017, the project gathered more than 9900 stargazers, 1100 contributors and was forked more than 3900 times. At the time of writing, more than 5500 issues are still open (more than 21200 closed overall) together with more than 1100 pull requests (almost 33000 closed overall)[52].

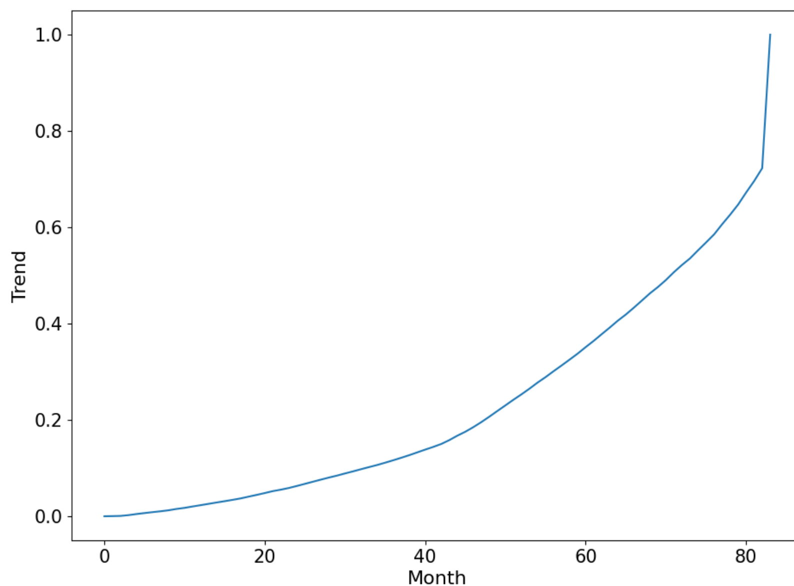

Figure 4.16: Merged metrics curve for the WordPress/gutenberg project. Source: Thesis author

From the merged curve in figure 4.16, it can be seen how the growth pace increases after the first two years of the project's lifetime. The final part of the curve shows a steep increase, which is probably the result of stronger growth in several metrics. From table 4.16, it appears that the number of *commits*, *issues*, *pull_requests* and changes *size* all follow a *Steep* pattern, which supports the behavior of the merged curved. Starting from the Word-

[52] https://github.com/WordPress/gutenberg

Press Gutenberg Index[53], three of the most active posts related to development activities are inspected to see if their timeline corresponds to any of the break points in table 4.16.

	stargazers	releases	size	commits	contributors	deployments	issues	forks	pull_requests	workflows
02/2017										
11/2017	0.14									
09/2018	0.33	0.05		0.26				0.19		
02/2019							0.04			
05/2020	0.57									
03/2021		0.44		0.55	0.47					
01/2022	0.75				0.61			0.67		
06/2022				0.74						
11/2022		0.79								
04/2023			0.79	0.88			0.73		0.55	
01/2024	1	1	1	1	1	0	1	1	1	0

Table 4.16: Metrics patterns for the WordPress/gutenberg project

A post from January 2022 discussed the need to update the version in the JSON schema of the front-end theme files to align with WordPress 5.8[54]. This involved changes in the structure of the JSON files themselves as well as updates in CSS files. The discussion was resolved in April 2022. During this period, a shift in the growth rate of development-related metrics is expected. This is confirmed in table 4.16, where it can be seen that the number of *commits* go from a *Shallow* to a *Steep* pattern around June 2022.

On March 2021, a three-month plan was posted to prepare for the release of version 5.8 of the software[55]. The schedule involved the merge of opened requests, the fix of eventual bugs and the release of a beta version before the final one. In light of this, it is expected that the development-related metrics show pattern changes around the time of the release plan. From table 4.16, it can be seen how the number of *commits*, *releases* and *contributors* register a break point around March 2021. This reflects the previous assumptions based on the posted release plan. The same reasoning can be applied to the plan to prepare for WordPress 6.2.1 posted on May 2023[56]. The *Steep* pattern for the number of *commits*, *issues*, *pull_requests* and changes *size* between April 2023 and January 2024 reflect the need to introduce changes to align with the latest version of WordPress at the time. Overall,

[53] https://make.wordpress.org/core/handbook/references/keeping-up-with-gutenberg-index/

[54] https://make.wordpress.org/core/2022/01/08/updates-for-settings-styles-and-theme-json/

[55] https://make.wordpress.org/core/2021/03/30/5-8-pre-planning/

[56] https://make.wordpress.org/core/2023/05/03/wordpress-6-2-1-planning/

the break points inspection for this project led to successful findings of real events to link to pattern changes.

saltstack/salt Salt is an event-driven framework to set up and manage cloud systems to ensure state consistency across all the deployed components. Started in early 2011, the project gathered more than 13000 stargazers, 2500 contributors and was forked more than 5000 times. At the time of writing, more than 2000 issues are still open (more than 23000 closed overall) together with more than 200 pull requests (almost 40000 closed overall)[57].

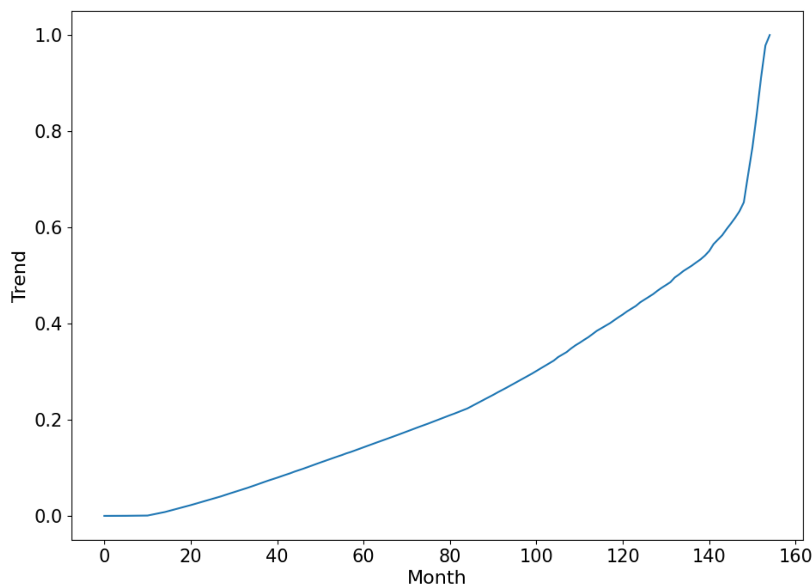

Figure 4.17: Merged metrics curve for the saltstack/salt project. Source: Thesis author

The merged curve of all the metrics trends is illustrated in figure 4.17 and the summary of all the metrics patterns are shown in table 4.17. From the merged curve, it can be seen how the overall metrics of the project follow a constant growth, starting slow, picking up in the middle and rising fast in more recent months. This is reflected in the patterns representation of the curve, which starts with a *Shallow* growth before alternating *Steep* and *Shallow* patterns. Starting from the Salt Project security announcement forum[58], several posts are inspected to see if their timeline corresponds to any of the break points in table 4.17.

In the period between January 2021 and September 2021, a series of security update

[57] https://github.com/saltstack/salt
[58] https://saltproject.io/security-announcements

	stargazers	releases	size	commits	contributors	deployments	issues	forks	pull_requests	workflows
02/2011										
07/2013		0.22								
12/2013	0.14									
10/2014				0.45						
03/2015					0.39			0.29		
08/2015		0.4								
01/2016								0.39		
04/2017	0.49									
09/2017							0.07			
02/2018	0.65			0.76	0.75					
07/2018								0.66		
12/2018										
03/2020							0.35		0.04	
08/2020		0.78								
02/2023			0.91			0	0.78		0.4	0
12/2023	1	1	1	1	1	1	1	1	1	1

Table 4.17: Metrics patterns for the saltstack/salt project

releases were announced[59,60,61,62,63] to inform users and contributors about fixes applied to prevent detected vulnerabilities. It is expected that the growth patterns of development-related metrics would be affected by such a series of releases in less than a year. From table 4.17, it can be seen how the number of *issues*, *pull_requests* and the changes *size* register a break point around February 2023, which is a year after the security updates were released. This can indicate users and contributors raising additional issues in light of the latest changes.

An additional reason for the pattern shift in the metrics discussed above can be the critical vulnerability announcement posted on June 2022[64]. As the development team suggested users to apply the fixes to their projects, it is possible to expect a rise in the number of *issues* and *pull_requests* as eventual new vulnerabilities are identified or older ones are still applicable. The same reasoning can be related to two additional vulnerability announcements

[59] https://saltproject.io/security-announcements/2021-01-29-advisory/
[60] https://saltproject.io/security-announcements/2021-02-04-advisory/
[61] https://saltproject.io/security-announcements/2021-02-25-advisory-01/
[62] https://saltproject.io/security-announcements/2021-02-25-advisory-02/
[63] https://saltproject.io/security-announcements/2021-09-21-advisory/
[64] https://saltproject.io/security-announcements/2022-06-13-advisory/

from August 2023[65] and January 2024[66]. In this case, table 4.17 shows how the number of *issues*, *pull_requests* and changes *size* all follow a *Steep* growth pattern, which can be related to both users reporting new bugs and the code changes to implement the fixes. Overall, the break points inspection for this project led to successful findings of real events to link to pattern changes.

ansible/awx AWX is a web-based interface that enables a user-friendly interaction with Ansible to deploy and manage cloud services. Started in mid-2017, the project gathered more than 13500 stargazers, 450 contributors and was forked more than 3000 times. At the time of writing, more than 1600 issues are still open (more than 6500 closed overall) together with more than 100 pull requests (almost 6900 closed overall)[67].

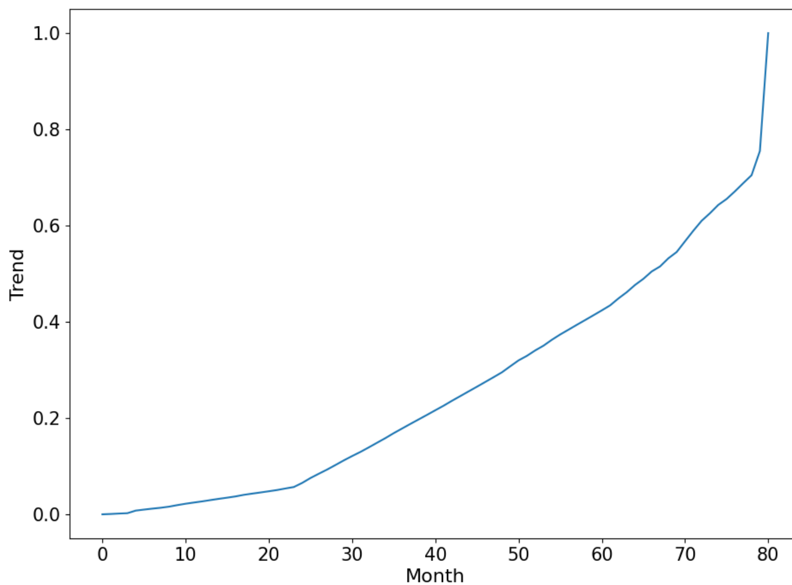

Figure 4.18: Merged metrics curve for the ansible/awx project. Source: Thesis author

The merged curve in figure 4.18 shows how the project development picked up momentum after a couple of years of slower growth in the beginning. The growth rate also shows a strong rise in the past year, which is also reflected by the amount of *Steep* patterns in table 4.18 in the period between February 2023 and January 2024. Starting from the Ansible AWX discussion forum[68], a set of popular posts are analyzed to check if their timeline corresponds to any of the break points in table 4.18.

[65] `https://saltproject.io/security-announcements/2023-08-09-advisory/`
[66] `https://saltproject.io/security-announcements/2024-01-30-advisory/`
[67] `https://github.com/ansible/awx`
[68] `https://forum.ansible.com/`

	stargazers	releases	size	commits	contributors	deployments	issues	forks	pull_requests	workflows
05/2017										
02/2018				0.19						
05/2019	0.44				0.31			0.35		
03/2020				0.62			0.28			
01/2021		0.1								
06/2021							0.52			
02/2023		0.73	0.82		0.77		0.83		0.4	
01/2024	1	1	1	1	1	0	1	1	1	0

Table 4.18: Metrics patterns for the ansible/awx project

A series of releases announcements between September 2023 and December 2023[69,70,71,72,73,74] is expected to introduce a stronger growth for the number of *releases* and other development related metrics. This is reflected by the *Steep* growth pattern for the number of *releases*, *issues* and changes *size* in table 4.18. Additionally, the pattern related to the former two metrics can also be linked to the introduction of a new feature that allows AWX to be better integrated into Kubernetes[75]. As the implementation for this feature spanned between August 2023 and April 2024, it is expected to see a shift in the growth of the number of *issues* and changes *size* after the February 2023 break point shown in table 4.18. The break points inspection for this project led to successful findings of real events to link to pattern changes.

Overall, both approaches followed to validate the break point detected in the time series segmentation process brought encouraging results as it was possible to identify significant events in the vicinity of the detected break points dates. Only 3 of the 10 selected projects (PatternFly-React, pip, CockroachDB) were not linked to real events beyond GitHub. Therefore, the discussed validation process fulfilled its aim to verify the reliability of the break points detection described in chapter 3.3.1. Additionally, from the inspections, a dataset of real world events for the analyzed projects has been built as a contribution to future work.

[69] https://forum.ansible.com/t/release-annoucement-awx-v23-1-0/611
[70] https://forum.ansible.com/t/announcing-awx-23-2-0-and-awx-operator-2-6-0/1132
[71] https://forum.ansible.com/t/announcing-awx-23-3-0-and-awx-operator-2-7-0/1610
[72] https://forum.ansible.com/t/announcing-awx-23-3-1-and-awx-operator-2-7-1/1927
[73] https://forum.ansible.com/t/announcing-awx-23-4-0-and-awx-operator-2-7-2/2228
[74] https://forum.ansible.com/t/announcing-awx-23-5-1-and-awx-operator-2-9-0/2763
[75] https://forum.ansible.com/t/ability-to-allow-inbound-connection-to-awx-receptor-mesh-on-kubernetes/215

4.3.2 RQ3.2: Does the Metrics Selection Need to be Extended?

Following the findings from research question 3.1, 3 of the 10 selected projects (PatternFly-React, pip, CockroachDB) did not register a detection of real events beyond GitHub given their break points derived from the metrics time series segmentation. Therefore, this research question aims to assess if extending the metric selection would help identify new break points that can be linked to real events in the project's history. The additional metrics that are considered in this analysis are picked from the ones that were omitted in chapter 3.2.1 due to their non-trivial retrieval from the GitHub API:

- **Licensing**: Open-source software licenses allow the content to be used, edited and distributed to different extents. Therefore, a change in licensing might reflect on the community involvement in the development of a project.
- **Complexity**: The complexity of software describes a set of properties of the code base, which can be represented by several metrics. In this case, the cyclomatic and cognitive complexity are chosen as they are available to collect through software analysis tools. In particular, the cyclomatic complexity computes the number of linearly independent paths in a program. A lower score means that the code is easier to test and modifying it implies fewer risks [29]. The cognitive complexity reflects how easy it is to read and understand the code, with lower scores being good outcomes [64]. Both scores data have been collected as monotonic time series to be consistent with the data collection from chapter 3.2.1.

For each one of the 3 projects, the licensing changes have been collected through the license files' commits history on GitHub. The cyclomatic and cognitive complexity data have been collected by loading a sample of 20 previous versions of the code-base to SonarCloud[76] for each project. SonarCloud is a tool that allows the analysis of code and outputs many quality metrics, including the above-mentioned ones for complexity. For each project, the two metrics time series have been segmented and mapped to evolution patterns following the same procedure of research question 3.1 (4.3.1).

patternfly/patternfly-react In March 2019, a change of license from Apache2.0 to MIT has been detected in the related file on GitHub[77]. From table 4.9 in chapter 4.3.1, it can noticed how the number of forks has a shift in the growth rate despite still holding a *Steep* pattern. The license change might have also been dictated by the surge of forks from mid-2018, which could have influenced the main contributors to shift to the more permissive MIT license type.

The evolution patterns of the cyclomatic and cognitive complexity are shown in table 4.19. Both metrics follow a *Shallow* growth without shifts in pattern type. The break point detected in July 2022 was already present in table 4.9 and did not lead to any real events. Therefore, the addition of the complexity metrics did not contribute in this regard.

[76] `https://www.sonarsource.com/products/sonarcloud/`
[77] `https://github.com/patternfly/patternfly-react/pull/1456`

	cyclomatic_complexity	**cognitive_complexity**
03/2017		
07/2022	0.39	0.39
03/2024	1	1

Table 4.19: Complexity metrics patterns for the patternfly/patternfly-react project

pypa/pip Since the project's creation, the MIT license has not changed. The only relevant editing was the removal of the CA (certificate authority) certificates notice related to one dependency as it was updated in October 2013[78]. However, the absence of major changes in the license type did not provide additional break points.

	cyclomatic_complexity	**cognitive_complexity**
03/2011		
02/2021	0.73	0.74
01/2024	1	1

Table 4.20: Complexity metrics patterns for the pypa/pip project

The complexity metrics both followed a *Shallow* pattern with a shift in the growth rate in February 2021. This break point did not lead to a pattern change and it was already detected in table 4.12. Thus, the analysis of the cyclomatic and cognitive complexity did not contribute to linking the detected break points to more real events.

cockroachdb/cockroach After starting with the Apache2.0 license in 2014, a major change was adopted in June 2019 with the introduction of the stricter BSL (business source license) license[79,80]. However, this change does not seem to have affected the growth patterns of other metrics as shown in table 4.11 from chapter 4.3.1.

	cyclomatic_complexity	**cognitive_complexity**
02/2014		
08/2021	0.16	0.14
05/2022	0.51	0.48
03/2023	0.8	0.79
01/2024	1	1

Table 4.21: Complexity metrics patterns for the cockroachdb/cockroach project

[78] https://github.com/pypa/pip/pull/1256
[79] https://www.cockroachlabs.com/blog/oss-relicensing-cockroachdb/
[80] https://news.ycombinator.com/item?id=20097077

From table 4.21, it can be seen how around May 2022, a shift in the growth pattern of both complexity metrics occurred by going from *Shallow* to *Steep*. This break point was not detected in the analysis conducted in chapter 4.3.1 and it can be linked to a series of releases that occurred around that time. In April 2022 the beta versions of a major release of the software started to come out[81]. This release focused on refactoring the pipeline required to set up a Cockroach database cluster, which led to more stable releases starting in May 2022. For this project, the analysis of the complexity metrics successfully led to finding real events to link to newly detected break points.

Overall, the additional inspections conducted on the 3 projects with missing real events links from research question 3.1 (4.3.1) brought mixed results. While the extended metrics selection led to the identification of relevant licensing events for two out of the three, the complexity metrics inspection was useful only for a single project. While these results are related to how projects are managed and how much information is made available on them, it is worth stressing the importance of the metrics selection process as adding more metrics might result in further insights about the evolution of open-source software.

[81] https://www.cockroachlabs.com/blog/cockroachdb-22-1-release/

Chapter 5

Results Discussion, Limitations and Future Work

The framework presented in this work followed a set of approaches based on previous literature that have been applied to the domain of open-source software evolution. Therefore, it is important to contextualize the results that can be derived from this framework and to discuss its shortcomings and possible next steps to improve it. The methodologies insights are discussed in section 5.1. Section 5.2 reflects on the patterns prediction and its effectiveness based on the known historical data. The break points detection and their linking to real events are discussed in section 5.3. Finally the threats to validity are listed in section 5.4.

5.1 Framework Insights

Patterns Modeling and Repositories Clustering The generalized evolution patterns shown in chapter 4.1.1 suggest which type of growth open-source projects go through based on their metrics time series data. The *Steep*, *Shallow* and *Plateau* patterns identify trends that relate to the input data and can be used to represent heterogeneous measures coherently. Comparing different metrics through their evolution patterns provides a mean to investigate their relationship and influence on each other. The break points that delimit these patterns represent the points in time at which a growth shift occurred. This can guide investigations about the causes that triggered a change in the evolution pattern. Additionally, the patterns sequences are directly linked to the statistical properties of the original time series data. This representation enables further comparisons between projects, which has been discussed with the results of the multivariate time series clustering.

Although these observations provide confidence about the patterns modeling and clustering methodology, it must be remarked that the obtained results are directly related to the nature of the input data. In the case of this work, the metrics time series followed a non-decreasing monotonic behavior, which explains why the patterns polynomials represent increasing growth only. Additionally, the size of the dataset can also be linked to the number of curves and clusters obtained at the end, with more data potentially resulting in more of both. Therefore, researchers must have a clear understanding of the type of input

data and its behavior before they derive the generalized evolution patterns.

Finally, despite that the methodology was built on previous literature backed by evaluation experiments, what is still missing is the direct involvement of researchers. This additional validation could bring a better perspective on how actionable the framework is and how it can be improved to make it more robust and useful. In this regard, a survey can be organized by preparing a set of hypothetical evolution scenarios for a set of open-source projects. The interviewees can then give feedback based on their experience on working on such projects. This would both support the proposed methodologies and also gather further insights on how the framework can be extended.

Forecasting Models The results presented for research question 1.2 in chapter 4.1.2 obtained using the multivariate time series forecasting models can be interpreted based on the prediction goals. While the bottom line of the experiments highlighted how more known historical training data leads to better results, the outcomes themselves need further discussion. The scores linked to the patterns prediction (*Performance, Deviation*) show good output even for the smallest amount of known history for the metric in question. This can be related to the fact that the patterns classifier is trained on highly generalized curves. Thus, even if the predicted segments do not resemble the original data well (with R^2 barely over 0.5), there is still a high chance that they still fall under the same growth pattern. This means that researchers can use the forecasting model for patterns prediction even with limited historical data available for a project. However, they must keep in mind that this situation would not favor the forecasting of accurate future values for the metrics, which requires more training data.

The break points detection related scores (*Precision, Recall, Random Index, Hausdroff*) also improve as more training data is used. Although the patterns prediction results are still acceptable for small known data, the break points detection struggles to find alignment between the break points from the predicted curves and the original ones. As the model does not fit the training data well, the accuracy of the forecasts does not resemble the original time series. As suggested by the *Hausdroff* score, the months mismatch can be large for some of the predicted metrics break points with a difference of over 20 months. As the sliding window method discussed in chapter 3.3.1 marks break points following shifts in the windows averages, it might occur that the predicted time series values do not present the same behavior around the actual break points. This can lead to missing break points and/or them being identified at later months. This consequently impacts the *Precision*, *Recall* and *Random Index* scores. Therefore, it can be worth conducting further studies focusing on comparing results obtained with different window sizes and trying to find the optimal one.

Additionally, an idea for further improvements and research can be the exploration and comparison of different forecasting models applied to multivariate time series. In the presented methodology, XGBoost was used to train the prediction models and it obtained encouraging results. The choice of this model was also dictated by the available computing resources and time, which required a reasonably fast and light model. However, discussion in previous literature highlighted how neural networks are being involved in this research domain. Therefore, it is worth conducting more studies to understand which methodology

brings the most advantages to multivariate time series forecasting in the context of open-source software evolution.

Metrics Importance The discussion of research question 1.3 in chapter 4.1.3 gave an overview of how the metrics affect the forecasting models results and how they play a role in each others' evolution. On this note, further discussions can be conducted by following table 4.4. From the values, it can be seen how the growth of commits affects the stargazers and the size of a project. This relationship can be related to the fact that more commits are the result of more contributors involved that might provide stargazers to the project. On the other hand, the commits introduce changes in the code base and affect its size through additions and deletions. The mutual importance between the pull requests and the number of issues is expected as the former is usually a consequence of the latter. The link between workflows and pull requests can be explained by how the project setup can trigger specific actions based on development activities. It is also worth noticing how the number of forks relates to the number of pull requests, which can be seen as bug fixes in the parallel developments being applied to the main project. These insights can give researchers a clearer picture of the dynamics within open-source projects evolution and allow them to focus on which metrics have the greatest impact. The discussed relationships between metrics and their influence on each other can become a topic to follow in future research, where multivariate forecasting models can be used to evaluate the changes in growth for a metric based on input data with different growth patterns. Additionally, future works can also study how the metrics growths and mutual importance are linked to the project type to potentially derive new measures of similarity.

5.2 Patterns Prediction

The results from the evaluation of research question 2 in chapter 4.2 showed how the number of known patterns affects the future patterns prediction. For the forecast of just the patterns sequence, repeating the latest known pattern type already yields good enough results even for a very small amount of available history (2 months). On the other hand, correctly predicting a change in pattern would require a better knowledge of previous ones ($>$ 12 months). As for the experiment models, the baseline one based on pattern repetition proved to be almost as performant as the more refined N-grams based one. However, the information provided by the N-grams computation can be used to derive further insights about the evolution of open-source projects. In particular, researchers can contextualize these results in the theoretical evolution phases discussed by previous literature in chapter 2.2 to provide additional proof about the phases that an open-source project has been through.

From table 4.6, it can be seen how likely it is that the identified patterns occur through the history of the gathered metrics data. As pattern repetition is the most likable scenario, focusing on pattern changes can drive several reflections. The *Plateau* has the highest chance of showing as the first pattern. This is because many metrics show no growth in the first stages of their history. The *Steep* and *Shallow* growths have almost the same probability of being the first pattern, which can be seen as a phase of initial development for an open-

source project. The *Plateau* pattern also shows itself as the most likely to appear after a shift from the *Shallow* and *Steep* patterns. This might be linked with the related software project reaching maturity, which would then require little maintenance. The *Shallow* pattern shows a good probability of appearing after a *Steep* one. This can represent a support phase where bugs are fixed and small improvements are made in the project after a sizeable amount of changes are introduced. In the light of this discussion, the theoretical framework of the software evolution phase described by previous literature in chapter 2.2 can be linked to the tangible evolution patterns found in this step of the methodology as follows:

Initial Development : *Steep* or *Shallow* pattern as first phase.

Maintenance : Sequence of multiple *Shallow* patterns in the middle phases, mostly following a *Steep* phase. This can occur cyclically in the history of open-source software.

Evolution : Presence of a *Steep* pattern as a middle phase. This can also happen cyclically in the history of open-source software.

Servicing/Legacy Support : Sequence of multiple *Shallow* patterns in the middle phases, which can be interleaved by *Plateau* patterns. This occurs more as a later stage but can also repeat itself after more *Evolution* and *Maintenance* phases occur.

Maturity/Archiving : Sequence of *Plateau* patterns only that appear at the last stages in the history of open-source software.

As the framework provides the tools to allow researchers to enhance time series analysis in the context of open-source software evolution, it is advised that further research will also focus on how these approaches can be used to link their empirical results with the existing theories about the evolution phases of software projects.

5.3 Break Points Detection and Real Events

Results from chapters 4.3.1 and 4.3.2 showed how the time series segmentation methodology provides a set of break points that can be linked to real events that occurred in the projects' lifetime. This inspection highlighted a use case for the framework that can assist researchers in finding relevant periods in open-source evolution more efficiently. By knowing when significant growth shifts occurred and which metrics are linked to them, it can be possible to have a clearer idea of how to search for the causes of such changes. However, it must be remarked that for some projects no events were found following the indicated break points. Although from the analyzed repositories it emerged that each one follows different ways of handling announcements, issues tracking and releases change-logs, eventual shortcomings of the used segmentation methodology must be discussed.

As mentioned in chapter 5.1, the sliding window approach used to segment time series might have caused the misalignment and the missed detection of break points between predicted and original data due to the window size (12 months). Although this case does not concern forecast curves, the possibility that some break points have been overlooked

should not be excluded. As this can have an impact on the study of open-source software evolution by making it harder to link the results to real world events, further studies are again recommended to try different window size settings in the methodology to find the optimal one. Additionally, a dataset of events linked to changes in growth in the metrics evolution of open-source software should also be introduced and kept up to date. This will allow researchers to easily query the data they need and eventually enrich them with more findings from external resources. As a contribution, a dataset of over 30 events related to the inspected projects has been built and openly shared to serve as a starting point for future works.

Considerations on the metrics selection can also be made as research question 3.2, in chapter 4.3.2, assessed if more measures would lead to additional events linked to break points. In the conducted research, the metrics selection was derived from the factors extracted from previous literature on software evolution, which resulted in the time series data collection of 10 metrics for each open-source project in the dataset. However, as described above, the inability to find the correspondence between break points and real events for some repositories led to the investigation of additional metrics, which produced mixed results. On this note, it is important to underline that the data collection described in this thesis relies on the GitHub API only. Although this source offers a remarkably wide selection of historical data from open-source software, it is also true that it is difficult to use for collecting specific metrics. This was shown in chapter 4.3.2, where the complexity metrics were generated and collected using SonarCloud. Given the available time and resources, the GitHub API was the most optimal solution to carry out the current work and still represents a great source of information.

Further research is advised to look deeper into how open-source software time series data can be retrieved and translated to usable metrics to study a project's evolution. Studies should also focus on which metrics selection is most suited for specific purposes so researchers can better orient themselves when analyzing a certain domain of open-source software evolution. Additionally, focusing on how the number of break points affects the metrics growth can lead to further findings about differences and similarities between projects' evolution.

5.4 Threats to Validity

The construct validity is challenged by the metrics selected to represent the open-source projects using time series data. The threat has been mitigated by building the selection process on top of an extensive literature review to include the most relevant measures identified by the past 20 years of research, as discussed in chapter 3.2.1. However, these metrics needed to be contextualized to an up-to-date open-source software data source, which was identified as the GitHub API. Despite most of the metrics being easily retrievable, the best practices adoption and the project size needed further interpretation. Still following previous work [14], the best practices adoption was linked to the presence of CI/CD practices in the repositories, which were provided by the GitHub API. On the other hand, the size metric collection incurred limitations from the data source itself and a monthly down-sampling

was required to complete the task with the available resources. Although the data gathered for these metrics contributed to deriving insights from the results of chapter 4, further validation is recommended to establish a robust approach to translate abstract factors to tangible and retrievable metrics.

In terms of internal validity, the main threats are the selected time series approaches from different domains, discussed in chapter 3.3, applied to the analysis of open-source software evolution. This was mitigated by selecting the methodologies through literature reviews on the topics of patterns modeling, time series clustering and time series forecasting. The techniques were chosen also based on their flexibility and ease of applicability to other contexts beyond the ones they were experimented on. Additionally, the techniques were applied following the described setup as closely as possible given the available resources. Although the discussions in sections 5.2 and 5.3 highlighted eventual limitations and future recommendations, the followed methodologies produced results that reflected their scalability and reproducibility, as shown in chapter 4.

The external validity is threatened by the selected sample of open-source projects from GitHub, which can compromise the generalizability of the results. As discussed in chapters 3.2.1 and 3.2.2, GitHub has become the biggest host of open-source projects and is home to more than 100 million openly available repositories. The choice of the most popular open-source data source is in line with previous research [5][61][59], where the Sourceforge.net database was used as the most popular source at that time. In terms of the projects used for the data collection, previous work was also relied upon [14]. As explained in chapter 3.2.2, this gave the confidence of having a sample of more than 1300 projects with guaranteed activity in recent years. This amount of input data also suited the computational resources available. Additionally, the results from chapter 4 showed consistency when experiments were run with a variable input size giving confidence about the scalability of the framework. Furthermore, to determine whether these findings can still be applied to a larger sample of data, further validation is advised to be performed as discussed in section 5.1.

Chapter 6

Conclusions

This chapter aims to provide an overview of the contributions that resulted from this work in section 6.1 and a summary of the presented thesis in section 6.2.

6.1 Contributions

The work presented in this thesis aimed to provide a reproducible and scalable framework to support researchers in the analysis of the evolution of open-source software. The followed methodology allowed the creation of several models that can be leveraged in the study of the growth patterns of open-source projects. Overall, the contributions of this research are listed as follows:

1. Provided an extensive literature review on open-source software evolution that spans from 2000 to 2024.
2. Provided a literature review of multivariate time series segmentation, clustering and forecasting techniques from economics and healthcare domains that can be applied to open-source software evolution analysis.
3. Illustrated a detailed systematic literature review procedure to extract relevant open-source software evolution metrics from previous literature.
4. Built a dataset of 1328 open-source software time series metrics data by contextualizing the literature metrics to the GitHub API. The dataset is openly available at the following URL: `https://huggingface.co/datasets/MattiaBonfanti-CS/IN5000-MB-TUD-Dataset-MongoDB`.
5. Illustrated a methodology to segment, cluster and classify time series patterns with the definition of three generalized growth curves: *Steep; Shallow; Plateau*.
6. Trained a KNN classifier to assign one of the generalized curves to a new set of data. The model is openly available at the following URL: `https://huggingface.co/MattiaBonfanti-CS/IN5000-MB-TUD-Patterns`.

7. Illustrated an algorithm to merge time series curves based on their patterns sequence to get a generalized curve to describe a project's growth.

8. Described a methodology to cluster similar open-source software projects based on their metrics patterns sequences and the similarity between metrics pairs, which resulted in three clusters.

9. Trained a KNN classifier to assign new data to one of the defined clusters. The model is openly available at the following URL: `https://huggingface.co/MattiaBonfanti-CS/IN5000-MB-TUD-Clustering`.

10. Illustrated a methodology to forecast multivariate time series where metric predictions are based on the evolution of other metrics.

11. Trained 30 forecasting models (1 per metric for 3 clusters) to predict the growth of specific metrics based on the growth of other ones. The models are openly available at the following URL: `https://huggingface.co/MattiaBonfanti-CS/IN5000-MB-TUD-Forecasting`.

12. Provided a set of scripts to reproduce the methodology and to apply the steps of the support framework for open-source software projects. The code is openly available at the following URL: `https://github.com/IN5000-MB-TUD/data-analysis`.

13. Provided a simple API and UI to expose the functionalities of the scripts in a more user-friendly way. The code is openly available at the following URL: `https://github.com/IN5000-MB-TUD/data-app`.

14. Provided a dataset of over 30 events linked to detected break points for the 10 inspected open-source projects in chapter 4.3. The dataset is openly available at the following URL: `https://huggingface.co/datasets/MattiaBonfanti-CS/IN5000-MB-TUD-Real-Events-Dataset-MongoDB`.

The listed contributions provide the tools to allow researchers to analyze the evolution patterns of open-source software. Overall, the results of this thesis can be used as a starting point for further research in the field of open-source software evolution.

6.2 Conclusion

This thesis aimed to develop a reproducible and scalable framework to analyze the evolution of open-source software. The methodologies followed to create the framework involved the modeling, clustering and forecasting of time series data. An extensive inspection of previous research has been conducted to acquire the necessary understanding of open-source software evolution as well as multivariate time series techniques from different domains (economics and healthcare). A systematic literature review was then performed to extrapolate the most relevant metrics from papers over the past 20 years, which were then mapped to the current measures available in the GitHub API. The data collection process resulted

in the creation of a 1328 repositories dataset over the following metrics: *stargazers; open issues; open pull requests; commits; contributors; changes size; forks; deployments; successful workflows; releases*.

Time series segmentation was applied to split the metrics curves into segments, which were then clustered based on their statistical properties. The average segment coefficients were used to define three generalized growth patterns: *Steep; Shallow; Plateau*. These patterns were then used in the multivariate time series clustering, which relied on the metrics patterns sequences and metrics pairs similarities to describe open-source projects and to cluster them into three clusters. For each cluster, a set of 10 forecasting models (1 per metric) were trained to perform multivariate time series predictions for a specific metric based on the growth of other ones. The experiment results give confidence over the reproducibility and the scalability of the framework methodologies, which are tested over different sizes of input data in terms of the number of projects, the amount of historical data and the number of metrics. Additional inspections also suggest that the shifts in the metrics evolution patterns can be related to real events in a project's history.

Overall, this research fulfills its aim of developing a reproducible and scalable framework to analyze the evolution patterns of open-source projects. The described methodology and results can serve as a starting point for further research on the topic of open-source software evolution.

Bibliography

[1] Mark Aberdour. Achieving quality in open-source software. *IEEE Software*, 24: 58–64, 1 2007. ISSN 0740-7459. doi: 10.1109/MS.2007.2.

[2] Mamdouh Alenezi and Khaled Almustafa. Empirical analysis of the complexity evolution in open-source software systems. *International Journal of Hybrid Information Technology*, 8:257–266, 2 2015. ISSN 17389968. doi: 10.14257/ijhit.2015.8.2.24.

[3] Ahmed Baabad, Hazura Binti Zulzalil, Sa'adah Hassan, and Salmi Binti Baharom. Software architecture degradation in open source software: A systematic literature review. *IEEE Access*, 8:173681–173709, 2020. ISSN 2169-3536. doi: 10.1109/ACCESS.2020.3024671.

[4] Satanjeev Banerjee and Ted Pedersen. The design, implementation, and use of the ngram statistics package. In *Computational Linguistics and Intelligent Text Processing*, volume 2000, pages 370–381, 02 2003. ISBN 978-3-540-00532-2. doi: 10.1007/3-540-36456-0_38.

[5] E.J. Barry, Chris Kemerer, and Sandra Slaughter. On the uniformity of software evolution patterns. In *Proceedings of the 25th International Conference on Software Engineering*, pages 106– 113, 06 2003. ISBN 0-7695-1877-X. doi: 10.1109/ICSE.2003.1201192.

[6] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, page 73–87, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132530. doi: 10.1145/336512.336534.

[7] Omar Benomar, Hani Abdeen, Houari Sahraoui, Pierre Poulin, and Mohamed Aymen Saied. Detection of software evolution phases based on development activities. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, page 15–24. IEEE Press, 2015. doi: 10.5555/2820282.2820288.

[8] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. When and how to make breaking changes. *ACM Transactions on Software Engineering and Methodology*, 30:1–56, 10 2021. ISSN 1049-331X. doi: 10.1145/3447245.

[9] Angela Bonifati, Francesco Del Buono, Francesco Guerra, and Donato Tiano. Time2feat: learning interpretable representations for multivariate time series clustering. *Proc. VLDB Endow.*, 16(2):193–201, oct 2022. ISSN 2150-8097. doi: 10.14778/3565816.3565822.

[10] Hongyu Pei Breivold, Muhammad Aufeef Chauhan, and Muhammad Ali Babar. A systematic review of studies of open source software evolution. In *2010 Asia Pacific Software Engineering Conference*, pages 356–365. IEEE, 11 2010. ISBN 978-1-4244-8831-5. doi: 10.1109/APSEC.2010.48.

[11] Defu Cao, Yujing Wang, Juanyong Duan, Ce Zhang, Xia Zhu, Conguri Huang, Yunhai Tong, Bixiong Xu, Jing Bai, Jie Tong, and Qi Zhang. Spectral temporal graph neural network for multivariate time-series forecasting. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546. doi: 10.5555/3495724.3497215.

[12] Andrea Capiluppi, Jesús M. González-Barahona, Israel Herraiz, and Gregorio Robles. Adapting the "staged model for software evolution" to free/libre/open source software. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, page 79–82, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937223. doi: 10.1145/1294948.1294968.

[13] Andrea Capiluppi, Klaas-Jan Stol, and Cornelia Boldyreff. Exploring the role of commercial stakeholders in open source software evolution. In *Open Source Systems: Long-Term Sustainability*, volume 378, 09 2012. ISBN 978-3-642-33441-2. doi: 10.1007/978-3-642-33442-9_12.

[14] Debasish Chakroborti, Sristy Sumana Nath, Kevin A. Schneider, and Chanchal K. Roy. Release conventions of open-source software: An exploratory study. *Journal of Software: Evolution and Process*, 35, 1 2023. ISSN 2047-7473. doi: 10.1002/smr.2499.

[15] Cristian Challu, Kin G. Olivares, Boris N. Oreshkin, Federico Garza, Max Mergenthaler-Canseco, and Artur Dubrawski. N-hits: Neural hierarchical interpolation for time series forecasting. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, 1 2022. doi: 10.1609/aaai.v37i6.25854.

[16] Yen-Yu Chang, Fan-Yun Sun, Yueh-Hua Wu, and Shou-De Lin. A memory-network based solution for multivariate time-series forecasting. *ArXiv*, 9 2018. URL http://arxiv.org/abs/1809.02105.

[17] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016. URL http://arxiv.org/abs/1603.02754.

[18] Zhichao Chen, Leilei Ding, Zhixuan Chu, Yucheng Qi, Jianmin Huang, and Hao Wang. Monotonic neural ordinary differential equation: Time-series forecasting for cumulative data. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, CIKM '23, page 4523–4529, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701245. doi: 10.1145/3583780.3615487.

[19] Maximilian Christ, Nils Braun, Julius Neuffer, and Andreas Kempa-Liehr. Time series feature extraction on basis of scalable hypothesis tests (tsfresh – a python package). *Neurocomputing*, 307, 05 2018. doi: 10.1016/j.neucom.2018.03.067.

[20] Fu Lai Chung, Tak Chung Fu, Vincent Ng, and Robert W.P. Luk. An evolutionary approach to pattern-based time series segmentation. *IEEE Transactions on Evolutionary Computation*, 8:471–489, 10 2004. ISSN 1089778X. doi: 10.1109/TEVC.2004.832863.

[21] Kattiana Constantino, Fabiano Belém, and Eduardo Figueiredo. Dual analysis for helping developers to find collaborators based on co-changed files: An empirical study. *Software: Practice and Experience*, 53(6):1438–1464, 2023. doi: 10.1002/spe.3194.

[22] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. Free/libre open-source software development: What we know and what we do not know. *ACM Comput. Surv.*, 44(2), mar 2008. ISSN 0360-0300. doi: 10.1145/2089125.2089127.

[23] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. Free/libre open-source software development. *ACM Computing Surveys*, 44:1–35, 2 2012. ISSN 0360-0300. doi: 10.1145/2089125.2089127.

[24] Kareem A. Dawood, Khaironi Y. Sharif, Abdul A. Ghani, H. Zulzalil, A.A. Zaidan, and B.B. Zaidan. Towards a unified criteria model for usability evaluation in the context of open source software based on a fuzzy delphi method. *Information and Software Technology*, 130:106453, 2 2021. ISSN 09505849. doi: 10.1016/j.infsof.2020.106453.

[25] Lucas de Oliveira Prates. A more efficient algorithm to compute the rand index for change-point problems. *CoRR*, abs/2112.03738, 2021. URL https://arxiv.org/abs/2112.03738.

[26] André Luiz de Souza Guimarães, Helaine J. Korn, N. Shin, and Alan B. Eisner. The life cycle of open source software development communities. *Journal of Electronic Commerce Research*, 14:167, 2013. URL `https://api.semanticscholar.org/CorpusID:30555255`.

[27] Torgeir Dingsøyr, Sridhar Nerur, VenuGopal Balijepally, and Nils Moe. A decade of agile methodologies: Towards explaining agile software development. *Journal of Systems and Software*, 85:1213–1221, 06 2012. doi: 10.1016/j.jss.2012.02.033.

[28] Shengdong Du, Tianrui Li, Yan Yang, and Shi Jinn Horng. Multivariate time series forecasting via attention-based encoder–decoder framework. *Neurocomputing*, 388: 269–279, 5 2020. ISSN 18728286. doi: 10.1016/j.neucom.2019.12.118.

[29] Christof Ebert and James Cain. Cyclomatic complexity. *IEEE Software*, 33:27–29, 11 2016. doi: 10.1109/MS.2016.147.

[30] Dalson Figueiredo, Silva Júnior, and Enivaldo Rocha. What is r2 all about? *Leviathan-Cadernos de Pesquisa Polútica*, 3:60–68, 11 2011. doi: 10.11606/issn .2237-4485.lev.2011.132282.

[31] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33:725–743, 11 2007. ISSN 00985589. doi: 10.1109/TS E.2007.70731.

[32] R. Friedrich, S. Siegert, Joachim Peinke, Stephan Lück, Malte Siefert, Michael Lindemann, Jan Raethjen, Günther Deuschl, and Gerd Pfister. Extracting model equations from experimental data. *Physics Letters A*, 271:217–222, 06 2000. doi: 10.1016/S0375-9601(00)00334-0.

[33] Amir Hossein Ghapanchi. Investigating the interrelationships among success measures of open source software projects. *Journal of Organizational Computing and Electronic Commerce*, 25(1):28–46, 2015. doi: 10.1080/10919392.2015.990775.

[34] Michael Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings 2000 International Conference on Software Maintenance*, pages 131 – 142, 02 2000. ISBN 0-7695-0753-0. doi: 10.1109/ICSM.2000.883030.

[35] Michael Godfrey and Qiang Tu. Growth, evolution, and structural change in open source software. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, page 103. ACM Press, 2002. ISBN 1581135084. doi: 10.1145/ 602461.602482.

[36] Michael W. Godfrey and Daniel M. German. The past, present, and future of software evolution. In *2008 Frontiers of Software Maintenance*, pages 129–138, 2008. doi: 10.1109/FOSM.2008.4659256.

[37] Jesus M. Gonzalez-Barahona, Gregorio Robles, Israel Herraiz, and Felipe Ortega. Studying the laws of software evolution in a long-lived floss project. *Journal of Software: Evolution and Process*, 26(7):589–612, 2014. doi: 10.1002/smr.1615.

[38] Miguel Goulao, Nelson Fonte, Michel Wermelinger, and Fernando Brito e Abreu. Software evolution prediction using seasonal time analysis: A comparative study. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, pages 213–222, 2012. ISBN 9780769546667. doi: 10.1109/CSMR.2012.30.

[39] Miguel Goulão, Nelson Fonte, Michel Wermelinger, and Fernando Brito e Abreu. Software evolution prediction using seasonal time analysis: A comparative study. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, 03 2012. doi: 10.1109/CSMR.2012.30.

[40] David Guijo-Rubio, Antonio Durán-Rosal, Pedro Antonio Gutiérrez, Alicia Troncoso, and Cesar Martínez. Time-series clustering based on the characterization of segment typologies. *IEEE Transactions on Cybernetics*, PP:1–14, 01 2020. doi: 10.1109/TCYB.2019.2962584.

[41] Valery Guralnik and Jaideep Srivastava. Event detection from time series data. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 33–42. Association for Computing Machinery (ACM), 8 1999. doi: 10.1145/312129.312190.

[42] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance*, 18:207–236, 05 2006. doi: 10.1002/smr.325.

[43] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution*, 18:207–236, 5 2006. ISSN 1532060X. doi: 10.1002/smr.325.

[44] Jiawei Han, Wan Gong, and Yiwen Yin. Mining segment-wise periodic patterns in time-related databases. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, KDD'98, page 214–218. AAAI Press, 1998. doi: 10.5555/3000292.3000330.

[45] Michael R. Hoopmann, Veit Schwämmle, and Magnus Palmblad. 2023 special issue on software tools and resources: Accelerating research with new and evolving open source software. *Journal of Proteome Research*, 22:285–286, 2 2023. ISSN 1535-3893. doi: 10.1021/acs.jproteome.3c00033.

[46] Keisuke Hotta, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 73–82. ACM, 9 2010. ISBN 9781450301282. doi: 10.1145/1862372.1862390.

[47] Sanjay Jain, Habib A. Islam, Martin C. Goossen, and Anil Nair. Social movements and institutional entrepreneurship as facilitators of technology transition: The case of free/open-source software. *Research Policy*, 52:104672, 3 2023. ISSN 00487333. doi: 10.1016/j.respol.2022.104672.

[48] A. Jallow, M. Schilling, M. Backes, and S. Bugiel. Measuring the effects of stack overflow code snippet evolution on open-source software security. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 26–26, Los Alamitos, CA, USA, may 2024. IEEE Computer Society. doi: 10.1109/SP54263.2024.00022.

[49] Cun Ji, Mingsen Du, Yupeng Hu, Shijun Liu, Li Pan, and Xiangwei Zheng. Time series classification based on temporal features. *Applied Soft Computing*, 128:109494, 10 2022. ISSN 15684946. doi: 10.1016/j.asoc.2022.109494.

[50] Cun Ji, Mingsen Du, Yanxuan Wei, Yupeng Hu, Shijun Liu, Li Pan, and Xiangwei Zheng. Time series classification with random temporal features. *Journal of King Saud University - Computer and Information Sciences*, 35:101783, 10 2023. ISSN 13191578. doi: 10.1016/j.jksuci.2023.101783.

[51] Spencer S. Jones, R. Scott Evans, Todd L. Allen, Alun Thomas, Peter J. Haug, Shari J. Welch, and Gregory L. Snow. A multivariate time series approach to modeling and forecasting demand in the emergency department. *Journal of Biomedical Informatics*, 42:123–139, 2 2008. ISSN 15320464. doi: 10.1016/j.jbi.2008.05.003.

[52] Benymol Jose and Sajimon Abraham. Exploring the merits of nosql: A study based on mongodb. In *2017 International Conference on Networks & Advances in Computational Technologies (NetACT)*, pages 266–271, 2017. doi: 10.1109/NETACT.2017.8076778.

[53] Siim Karus and Harald Gall. A study of language usage evolution in open source software. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 13–22. ACM, 5 2011. ISBN 9781450305747. doi: 10.1145/1985441.1985447.

[54] Ahmed Kattan, Shaheen Fatima, and Muhammad Arif. Time-series event-based prediction: An unsupervised learning framework based on genetic programming. *Information Sciences*, 301:99–123, 4 2015. ISSN 00200255. doi: 10.1016/j.ins.2014.12.054.

[55] Rajdeep Kaur, Kuljit Kaur Chahal, and Munish Saini. Understanding community participation and engagement in open source software projects: A systematic mapping study. *Journal of King Saud University - Computer and Information Sciences*, 34:4607–4625, 7 2022. ISSN 13191578. doi: 10.1016/j.jksuci.2020.10.020.

[56] Chris F. Kemerer and Sandra Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25:493–509, 1999. ISSN 00985589. doi: 10.1109/32.799945.

[57] Eamonn J. Keogh and Michael J. Pazzani. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, KDD'98, page 239–243. AAAI Press, 1998. doi: 10.5555/3000292.3000335.

[58] Eamonn J. Keogh, Selina Chu, David Hart, and Michael J. Pazzani. An online algorithm for segmenting time series. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, ICDM '01, page 289–296, USA, 2001. IEEE Computer Society. ISBN 0769511198. doi: 10.5555/645496.657889.

[59] Murtaza Khan and Faizan Rehman. Free and open source software: Evolution, benefits and characteristics. *International Journal of Emerging Trends & Technology in Computer Science (IJETTCS)*, 1:1–7, 9 2012. URL `https://www.researchgate.net/publication/256088382_Free_and_Open_Source_Software_Evolution_Benefits_and_Characteristics`.

[60] Barbara Kitchenham and Pearl Brereton. A systematic review of systematic review process research in software engineering. *Information and Software Technology*, 55 (12):2049–2075, 2013. ISSN 0950-5849. doi: 10.1016/j.infsof.2013.07.010.

[61] Stefan Koch. Software evolution in open source projects—a large-scale investigation. *Journal of Software Maintenance and Evolution: Research and Practice*, 19:361–382, 11 2007. ISSN 1532-060X. doi: 10.1002/smr.348.

[62] Georg Von Krogh, Sebastian Spaeth, and Karim R. Lakhani. Community, joining, and specialization in open source software innovation: A case study. *Research Policy*, 32:1217–1241, 7 2003. ISSN 00487333. doi: 10.1016/S0048-7333(03)00050-7.

[63] Rakesh Kumar and Rinkaj Goyal. Modeling continuous security: A conceptual model for automated devsecops using open-source software over cloud (adoc). *Computers & Security*, 97:101967, 10 2020. ISSN 01674048. doi: 10.1016/j.cose.2020.101967.

[64] Luigi Lavazza, Abedallah Abualkishik, Geng Liu, and Sandro Morasca. An empirical evaluation of the "cognitive complexity" measure as a predictor of code understandability. *Journal of Systems and Software*, 197:111561, 11 2022. doi: 10.1016/j.jss.2022.111561.

[65] M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. doi: 10.1109/PROC.1980.11805.

[66] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Turski. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*, pages 20–32, 1997. doi: 10.1109/METRIC.1997.637156.

[67] Chunbin Lin, Etienne Boursier, and Yannis Papakonstantinou. Plato: approximate analytics over compressed time series with tight deterministic error guarantees. *Proc. VLDB Endow.*, 13(7):1105–1118, mar 2020. ISSN 2150-8097. doi: 10.14778/3384345.3384357.

[68] Junaid Maqsood, Iman Eshraghi, and Syed Sarmad Ali. Success or failure identification for github's open source projects. In *Proceedings of the 2017 International Conference on Management Engineering, Software Engineering and Service Sciences*, ICMSS '17, page 145–150, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348348. doi: 10.1145/3034950.3034957.

[69] Nora McDonald and Sean Goggins. Performance and participation in open source software on github. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '13, page 139–144, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319522. doi: 10.1145/2468356.2468382.

[70] Nadeem Mehmood, Rosario Culmone, and Leonardo Mostarda. Modeling temporal aspects of sensor data for mongodb nosql database. *Journal of Big Data*, 4, 03 2017. doi: 10.1186/s40537-017-0068-5.

[71] Aleksandr Mezhenin and Alena Zhigalova. Similarity analysis using hausdorff metrics. In *Majorov International Conference on Software Engineering and Computer Systems*, 2018. URL `https://api.semanticscholar.org/CorpusID:133597817`.

[72] Gabriela Karoline Michelon, David Obermann, Wesley K. G. Assunção, Lukas Linsbauer, Paul Grünbacher, Stefan Fischer, Roberto E. Lopez-Herrejon, and Alexander Egyed. Evolving software system families in space and time with feature revisions. *Empirical Software Engineering*, 27:112, 9 2022. ISSN 1382-3256. doi: 10.1007/s10664-021-10108-z.

[73] Vishal Midha and Prashant Palvia. Factors affecting the success of open source software. *J. Syst. Softw.*, 85(4):895–905, apr 2012. ISSN 0164-1212. doi: 10.1016/j.jss.2011.11.010.

[74] Arthur-Jozsef Molnar and Simona Motogna. A study of maintainability in evolving open-source software. *ArXiv*, abs/2009.00959, 2021. doi: 10.1007/978-3-030-70006-5_11.

[75] Elisa Yumi Nakagawa, Elaine Parros Machado de Sousa, Kiyoshi de Brito Murata, Gabriel de Faria Andery, Leonardo Bitencourt Morelli, and José Carlos Maldonado. Software architecture relevance in open source software evolution: A case study. In *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 1234–1239. IEEE, 2008. ISBN 978-0-7695-3262-2. doi: 10.1109/COMPSAC.2008.171.

[76] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE '02, page 76–85, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135459. doi: 10.1145/512035.512055.

[77] Kin G. Olivares, Cristian Challú, Federico Garza, Max Mergenthaler Canseco, and Artur Dubrawski. Machine learning forecast. PyCon Salt Lake City, Utah, US 2022, 2022. URL `https://nixtlaverse.nixtla.io/mlforecast/`.

[78] Iliana Paliari, Aikaterini Karanikola, and Sotiris Kotsiantis. A comparison of the optimized lstm, xgboost and arima in time series forecasting. In *2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA)*, pages 1–7, 07 2021. doi: 10.1109/IISA52424.2021.9555520.

[79] John Paparrizos and Luis Gravano. k-shape: Efficient and accurate clustering of time series. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1855–1870. ACM, 5 2015. ISBN 9781450327589. doi: 10.1145/2723372.2737793.

[80] J.W. Paulson, G. Succi, and A. Eberlein. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering*, 30: 246–256, 4 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.1274044.

[81] Maksym Petrenko, Denys Poshyvanyk, Vaclav Rajlich, and Joseph Buchta. Teaching software evolution in open source. *Computer*, 40:25–31, 11 2007. ISSN 0018-9162. doi: 10.1109/MC.2007.402.

[82] James Piggot and Chintan Amrit. How Healthy Is My Project? Open Source Project Attributes as Indicators of Success. In Etiel Petrinja, Giancarlo Succi, Nabil Ioini, and Alberto Sillitti, editors, *9th Open Source Software (OSS)*, volume AICT-404 of *Open Source Software: Quality Verification*, pages 30–44, Koper-Capodistria, Slovenia, June 2013. Springer. doi: 10.1007/978-3-642-38928-3_3.

[83] V.T. Rajlich and K.H. Bennett. A staged model for the software life cycle. *Computer*, 33(7):66–71, 2000. doi: 10.1109/2.869374.

[84] Nasir Rashid and Siffat Ullah Khan. Developing green and sustainable software using agile methods in global software development: Risk factors for vendors. In *Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering*, ENASE 2016, page 247–253, Setubal, PRT, 2016. SCITEPRESS - Science and Technology Publications, Lda. ISBN 9789897581892. doi: 10.5220/0005913802470253.

[85] Elyas Sabeti, Peter X.K. Song, and Alfred O. Hero. Pattern-based analysis of time series: Estimation. In *2020 IEEE International Symposium on Information Theory (ISIT)*, pages 1236–1241. IEEE, 6 2020. ISBN 978-1-7281-6432-8. doi: 10.1109/IS IT44484.2020.9174529.

[86] Walt Scacchi. Understanding open source software evolution: Applying, breaking, and rethinking the laws of software evolution, 2003. URL https://api.semanticscholar.org/CorpusID:2641835.

[87] Hamed Shariat Yazdi, Mahnaz Mirbolouki, Pit Pietsch, Timo Kehrer, and Udo Kelter. Analysis and prediction of design model evolution using time series. In Lazaros Iliadis, Michael Papazoglou, and Klaus Pohl, editors, *Advanced Information Systems Engineering Workshops*, pages 1–15, Cham, 2014. Springer International Publishing. doi: 10.1007/978-3-319-07869-4_1.

[88] Hamed Shariat Yazdi, Lefteris Angelis, Timo Kehrer, and Udo Kelter. A framework for capturing, statistically modeling and analyzing the evolution of software models. *Journal of Systems and Software*, 118:176–207, 2016. ISSN 0164-1212. doi: 10.1016/j.jss.2016.05.010.

[89] Jyoti Sheoran, Kelly Blincoe, Eirini Kalliamvakou, Daniela Damian, and Jordan Ell. Understanding "watchers" on github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 336–339, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328630. doi: 10.1145/2597073.2597114.

[90] Grigori Sidorov, Francisco Castillo, Efstathios Stamatatos, Alexander Gelbukh, and Liliana Chanona-Hernández. Syntactic n-grams as machine learning features for natural language processing. *Expert Systems with Applications: An International Journal*, 41:853–860, 02 2014. doi: 10.1016/j.eswa.2013.08.015.

[91] Miriam Sitienei, Argwings Ranyimbo, and Ayubu Okango. An application of k-nearest-neighbor regression in maize yield prediction. *Asian Journal of Probability and Statistics*, 24:1–10, 09 2023. doi: 10.9734/ajpas/2023/v24i4529.

[92] Mahbubul Syeed, Imed Hammouda, and Tarja Systa. Evolution of open source software projects: A systematic literature review. *Journal of Software*, 8:2815–2829, 11 2013. doi: 10.4304/jsw.8.11.2815-2829.

[93] Abhishek Tandon, Sharma Meera, Kumari Madhu, and Singh V.B. Entropy based software reliability growth modelling for open source software evolution. *Tehnicki vjesnik - Technical Gazette*, 27, 4 2020. ISSN 13303651. doi: 10.17559/TV-20181031061451.

[94] Kai Ming Ting. *Precision and Recall*, pages 781–781. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_652.

[95] Charles Truong, Laurent Oudre, and Nicolas Vayatis. A review of change point detection methods. *CoRR*, abs/1801.00718, 2018. URL http://arxiv.org/abs/1801.00718.

[96] Kimberly Truong, Courtney Miller, Bogdan Vasilescu, and Christian Kästner. The unsolvable problem or the unheard answer? a dataset of 24,669 open-source software conference talks. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 348–352, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3528488.

[97] Gert Valkenhoef, Tommi Tervonen, Bert Brock, and Douwe Postmus. Product and release planning practices for extreme programming. In *Agile Processes in Software Engineering and Extreme Programming*, volume 48, pages 238–243, 06 2010. ISBN 978-3-642-13053-3. doi: 10.1007/978-3-642-13054-0_25.

[98] Pham Vuong, Dat Trinh, Tieu Mai, Pham Uyen, and Pham Bao. Stock-price forecasting based on xgboost and lstm. *Computer Systems Science and Engineering*, 40: 237–246, 01 2022. doi: 10.32604/CSSE.2022.017685.

[99] Renzhuo Wan, Shuping Mei, Jun Wang, Min Liu, and Fan Yang. Multivariate temporal convolutional network: A deep neural networks approach for multivariate time series forecasting. *Electronics (Switzerland)*, 8, 8 2019. ISSN 20799292. doi: 10.3390/electronics8080876.

[100] Xiaozhe Wang, Kate Smith, and Rob Hyndman. Characteristic-based clustering for time series data. *Data Mining and Knowledge Discovery*, 13:335–364, 9 2006. ISSN 1384-5810. doi: 10.1007/s10618-005-0039-x.

[101] Yi Wang, Defeng Guo, and Huihui Shi. Measuring the evolution of open source software systems with their communities. *ACM SIGSOFT Software Engineering Notes*, 32:7, 11 2007. ISSN 0163-5948. doi: 10.1145/1317471.1317479.

[102] Jingwei Wu. *Open Source Software Evolution and Its Dynamics*. PhD thesis, University of Waterloo, CAN, 2006. URL `http://hdl.handle.net/10012/1095`.

[103] Zonghan Wu, Shirui Pan, Guodong Long, Jing Jiang, Xiaojun Chang, and Chengqi Zhang. Connecting the dots: Multivariate time series forecasting with graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 753–763. Association for Computing Machinery, 8 2020. ISBN 9781450379984. doi: 10.1145/3394486.3403118.

[104] Guowu Xie, Jianbo Chen, and Iulian Neamtiu. Towards a better understanding of software evolution: An empirical study on open source software. In *2009 IEEE International Conference on Software Maintenance*, volume 25, pages 51–60, 09 2009. doi: 10.1109/ICSM.2009.5306356.

[105] Jong Yoon, Jieun Lee, and Sung-Rim Kim. Trend similarity and prediction in time-series databases. *Proceedings of SPIE - The International Society for Optical Engineering*, 04 2000. doi: 10.1117/12.381734.

[106] Jierui Zhang, Liang Wang, Zhiwen Zheng, and Xianping Tao. Social community evolution analysis and visualization in open source software projects. In *Web Information Systems Engineering – WISE 2022*, page 38–45, Berlin, Heidelberg, 2022. Springer-Verlag. ISBN 978-3-031-20890-4. doi: 10.1007/978-3-031-20891-1_4.

[107] Lingyu Zhang, Wenjie Bian, Wenyi Qu, Liheng Tuo, and Yunhai Wang. Time series forecast of sales volume based on xgboost. *Journal of Physics: Conference Series*, 1873:012067, 04 2021. doi: 10.1088/1742-6596/1873/1/012067.

[108] Xiaohang Zhang, Jiaqi Liu, Yu Du, and Tingjie Lv. A novel clustering method on time series data. *Expert Systems with Applications*, 38:11891–11900, 9 2011. ISSN 09574174. doi: 10.1016/j.eswa.2011.03.081.

Appendix A

Database Schema

From the database schema in figure A.1, it is noticeable that some of the fields are listed as *Objects* type, which represents a dictionary of key-value pairs. More details on the inner structure of such *Objects* type fields are given below for each collection:

- ***repositories_data***
  - *releases*
    - key: release tag
    - value (Object): tag_name (String), target(String), body (string), draft (Boolean), preprelease (Boolean), created_at (Datetime), published_at (Datetime)
  - *workflows*
    - key: workflow unique id
    - value (Object): id (Integer), name (String), created_at (Datetime)
  - *environments*
    - key: environment unique id
    - value (Object): id (Integer), name (String), created_at (Datetime)
  - *languages*
    - key: programming language name
    - value (Integer): code bytes written in referred language
  - *branches*
    - key: branch unique id
    - value (Object): protected (Boolean)
  - *metadata*
    - key: database object actions
    - value (Object): created (Datetime), modified (Datetime)
- ***statistics_commits***

- *commits*
  * key: commit sha encoding
  * value (Object): author (String), date (Datetime)
- *contributors*
  * key: contributor unique username
  * value (Object): commits (Integer), first_commit (Datetime)

- ***statistics_deployments***
  - *deployments*
    * key: deployment unique id
    * value (Object): id (Integer), environment (String), transient_environment (Boolean), production_environment (Boolean), created_at (Datetime)
- ***statistics_forks***
  - *forks*
    * key: fork unique id
    * value (Object): id (Integer), number (Integer), state (String), title (String), user (String), comments (Integer), author_association (String), created_at (Datetime), updated_at (Datetime), closed_at (Datetime)
- ***statistics_pull_requests***
  - *pull_requests*
    * key: pull request unique id
    * value (Object): id (Integer), number (Integer), state (String), created_at (Datetime), merged_at (Datetime), closed_at (Datetime)
- ***statistics_workflow_runs***
  - *workflows*
    * key: workflow run unique id
    * value (Object): id (Integer), workflow_id (Integer), created_at (Datetime)
- ***statistics_size***
  - *size*
    * key: commit sha encoding
    * value (Object): additions (Integer), deletions (Integer)total (Integer, additions + deletions), size (Integer, additions - deletions), date (Datetime)

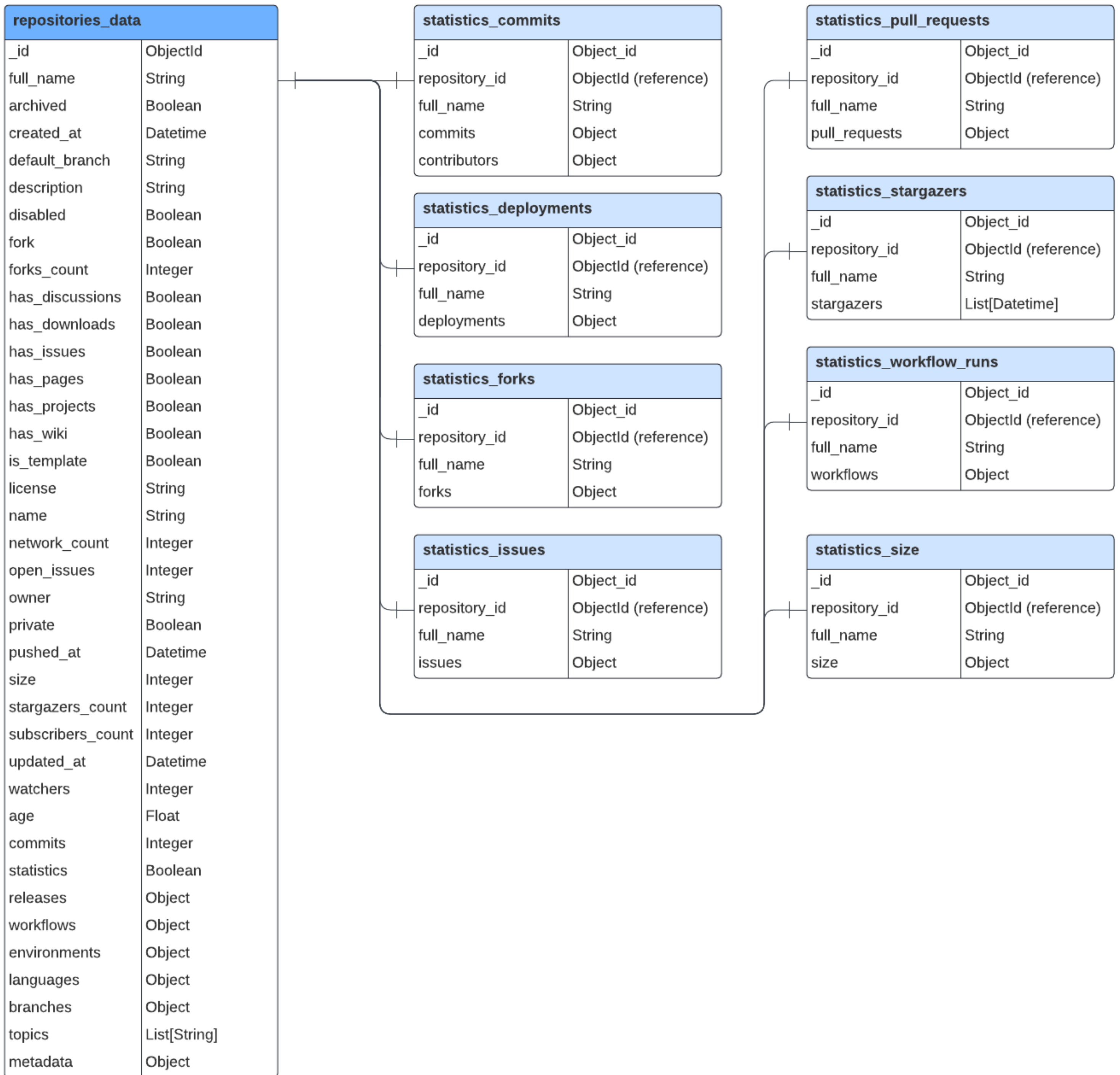

Figure A.1: MongoDB collections schema for repositories data and metrics time series data.
Source: Thesis author

Appendix B

Evolution Patterns Evaluation Plots

In this appendix, the generalized patterns curves obtained from the experiments discussed in chapter 4.1.1 are shown.

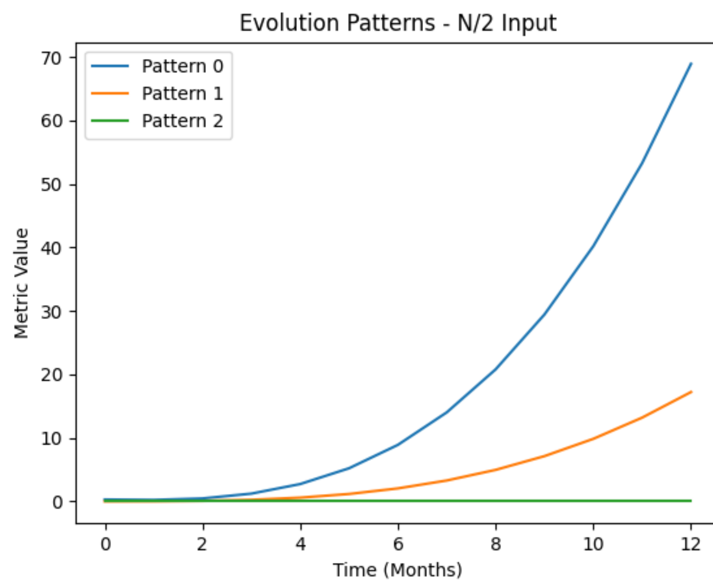

Figure B.1: Generalized time series segments growth patterns for N/2 (664 repositories) input. Source: Thesis author

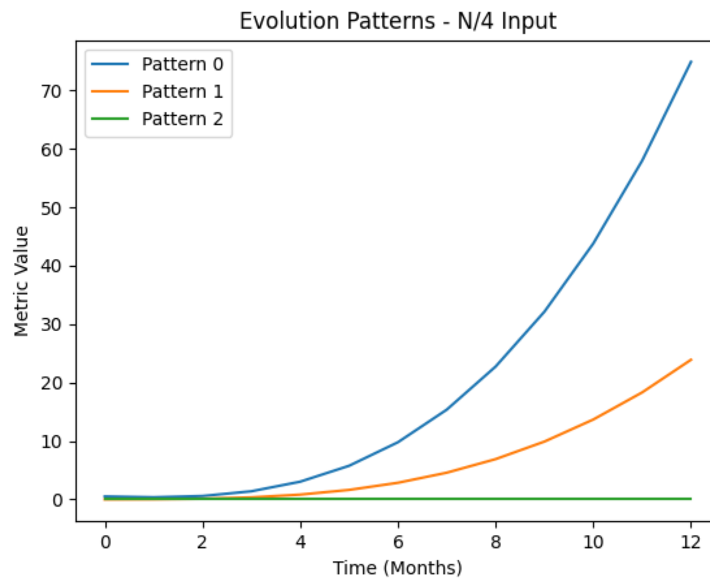

Figure B.2: Generalized time series segments growth patterns for N/4 (332 repositories) input. Source: Thesis author

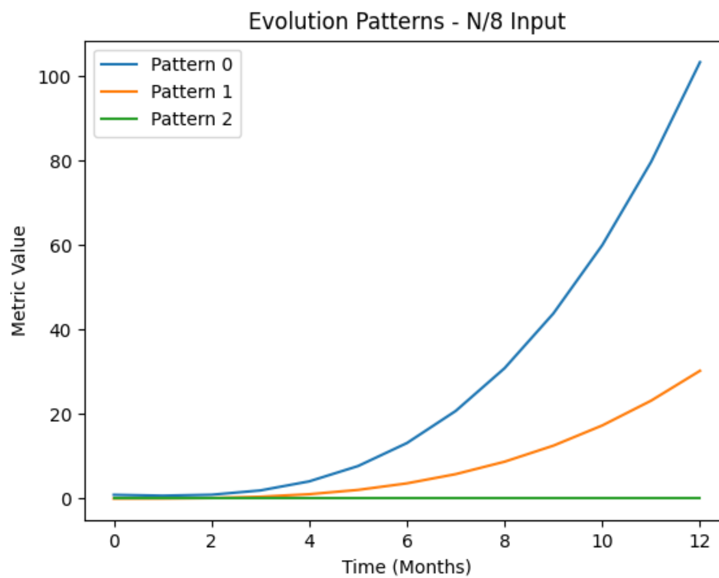

Figure B.3: Generalized time series segments growth patterns for N/8 (166 repositories) input. Source: Thesis author

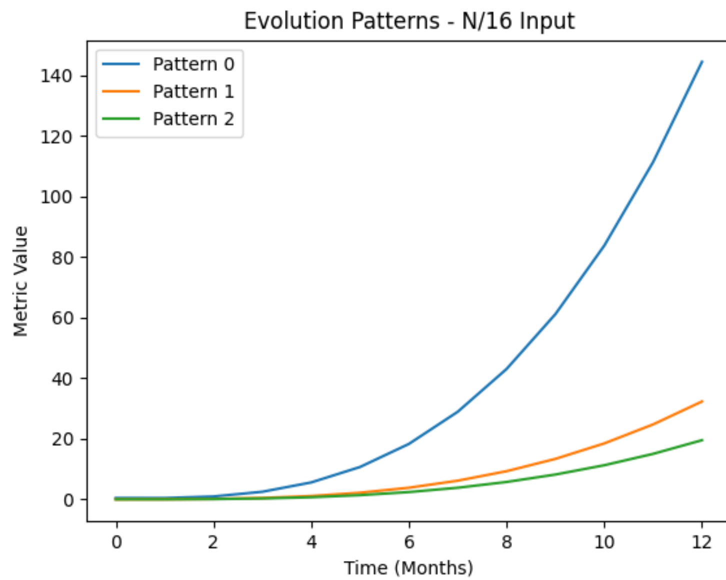

Figure B.4: Generalized time series segments growth patterns for N/16 (83 repositories) input. Source: Thesis author

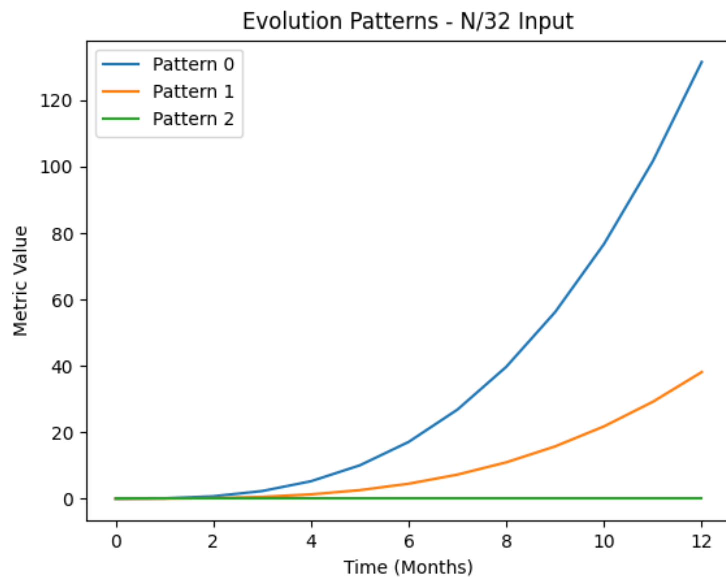

Figure B.5: Generalized time series segments growth patterns for N/32 (41 repositories) input. Source: Thesis author

Appendix C

Glossary

In this appendix an overview of used abbreviations is given.

OSS: Open-Source Software

FOSS: Free Open-Source Software

FLOSS: Free Libre Open-Source Software

GH: GitHub

KNN: K-nearest neighbors

MTS: Multivariate Time Series

MIT License: Massachusetts Institute of Technology License

BSL: Business Source License

CI/CD: Continuous Integration and Continuous Deployment

ARMA: Autoregressive Moving Average

ARIMA: Autoregressive Integrated Moving Average

API: Application Programming Interface

UI: User Interface