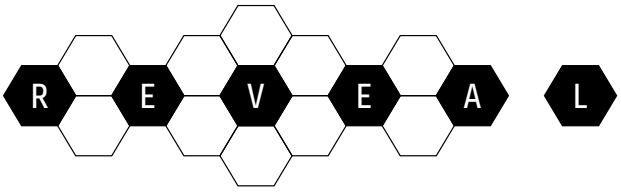


Università della Svizzera italiana

Software Institute

Mapping and Reifying the Software Documentation Landscape

Marco Raglianti

Research Advisor

Prof. Dr. Michele Lanza

Doctoral Dissertation submitted to the Faculty of Informatics of the Università della Svizzera italiana (USI)
in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Dissertation Committee

Gabriele Bavota	Università della Svizzera italiana, Switzerland
Carlo Alberto Furia	Università della Svizzera italiana, Switzerland
Gregorio Robles	Universidad Rey Juan Carlos, Spain
Federica Sarro	University College London, United Kingdom
Christoph Treude	Singapore Management University, Singapore

Dissertation accepted on 18 March 2025

Research Advisor
Prof. Dr. Michele Lanza

Ph.D. Program Co-Director
Prof. Walter Binder

Ph.D. Program Co-Director
Prof. Stefan Wolf

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Marco Raglianti
Lugano, 18 March 2025

To Ilaria,
the eye of the constant storm in my life
and the patient other-half throughout the last seventeen years

I live my life a quarter mile at a time,
and for those ten seconds or less,
I'm free

—Dominic Toretto

Abstract

Every software system (ideally) comes with one or more forms of documentation, a fundamental asset of every software project. Documentation is deemed useful for different stages of the software lifecycle (*e.g.,* design, implementation, maintenance) and is created and maintained by different stakeholders for different target audiences. In all this heterogeneity, multiple platforms (*e.g.,* instant messaging, Q&A websites, social networks) host and contribute a substantial amount of documentation. Developers resort to Discord, StackOverflow, and a mix of Google, emails, and Tweets to "get things done". We refer to this evolving heterogeneous collection of information sources as the *documentation landscape* of a software system. A systematic approach to discover, classify, and integrate these sources in a unifying framework is fundamental to understand the paradigm shifts and complex phenomena affecting modern software documentation.

To look at software documentation from a new perspective, we propose an approach to map and reify the documentation landscape of a software system. We identify and retrieve the sources of documentation deemed relevant by developers and investigate some of the many forms that documentation sources can take. We provide a holistic interpretation of the *software documentation landscape* and the phenomena determining its evolution. Traditional documentation is becoming a thing of the past, drifting towards fast (but also extremely volatile and noisy) communication channels, hosts of valuable chunks of the modern software documentation landscape.

We complement our approach with an evolutionary analysis that explores in detail software documentation in instant messaging applications and UML artifacts. Through a mix of object oriented domain modeling and software visualization techniques, we characterize documentation sources according to form and content, analyzing their interplay. Finally, we discuss the current limitations and the potential future of the software documentation landscape, highlighting the need of a machine-readable specification to fully automate our mapping approach.

Acknowledgements

First and foremost, I would like to thank my advisor, Michele Lanza, for giving me a chance, for seeing what others (sometimes including myself) could not see five years ago. Thanks for supporting me and giving me feedback, for teaching me how to be a better researcher, for being always direct and honest even if harsh sometimes. I really appreciated every discussion we had. I would like to extend my thanks to the REVEAL group, a living and breathing organism made of people but with its own identity, constantly shifting but always recognizable. Thanks to Csaba and Roberto, great co-supervisors in the early stages of my PhD, colleagues and friends towards the end of it (still ready to teach me a lesson when needed). Thanks Carmen, Stefano, and Mattia for keeping alive the tradition of a great research group in the next few years.

I would like to thank the Software Institute, USI, and the Dean's Office for being the best of containers for our activities. The support of brilliant professors, a solid institution, and an efficient infrastructure fostered my personal growth as a student, researcher, teaching assistant, and co-supervisor. A special mention goes to Andrea Mocci. I TA-ed your courses for so long that I feel part of the furnitures by now, but you were always available for a discussion, to provide advice or brainstorming power whenever I needed that. I tried to use it sparingly but you always found some time for my problems. Thank you.

Then, in *no particular order*, I would like to thank the many Bachelor, Master, and PhD students, the researchers and professors that enriched my experience with their different points of view and expertise. Thanks Adam, Adrian, Alberto, Alejandro, Alessandro, Andi, Andrea, Anthony, Antonio, Aron, Arthur, Avi, Bhagya, Bin, Breandan, CAF, Camilo, Cesare, Christoph, David, Davide, Deeksha, Denis, Divya, Edoardo, Emad, Emad, Fabio, Fabio, Federica, Francesco, Gabriele, Gianlo, Gregorio, Jaz, Jeferson, Jesper, Jin, Joe, Jun, Lanese, Linh, Luca, Marco, Mario, Mathieu, Matteo, Matteo, Matthias, Mauro, Maxime, Maxime, Mircea, Mohammad, Nicole, Olga, Olivier, Oussama, Paolo, Rosalia, Sara, Shinpei, Shushi, Sonia, Souhaila, Stefano, Susanna, Takashi, Tommaso, Valerie, Vincenzo, Vincenzo, Walid, Yasu, Yutaro, and all the others I surely forgot to mention.

Thanks to the Software Technology Lab at McGill University in Montreal, and in particular to Martin Robillard, for hosting me for three months last summer. You showed me a different but equally effective way of doing research so that I could compare and improve mine once back.

I would like to acknowledge with sincere gratitude the financial support of the Swiss National Science Foundation (SNSF) for the project "PROBE" (SNF Project No. 172799) and the SNSF and the Fonds de la Recherche Scientifique (F.R.S.-FNRS) for the joint Lead Agency project "INSTINCT" (SNF Project No. 190113) that supported me throughout the endeavors (and the many Star Alliance miles accumulated) in the last 4 years.

Finally, I would like to thank my family and friends. You have been part of this story for more than the last five years and you allowed me to be the person I am today.

Contents

List of Figures

List of Tables

Part I

Software Documentation

Context, History, and State of the Art

1

Introduction

Documentation is a fundamental aid for software comprehension, maintenance, and evolution [5]. In this chapter we introduce the context of software documentation and the massive changes it is experiencing under the pressure of emerging collaborative platforms and communication tools. We give an overview, from an initial definition of the *documentation landscape* to our central thesis statement. We conclude by detailing our conceptual and practical contributions towards a better understanding of modern software documentation and its evolution.

1.1 Classical and Modern Software Documentation

Software documentation has been studied extensively with respect to its quality and usefulness [6, 45, 54, 79, 83, 192, 231, 265]. The holy grail of software documentation is to "find the right information, in the right form, at the right time [253]". One way to do so is to provide developer documentation which is not only served, but also produced, on-demand, according to developer needs for the task at hand [193, 253]. We argue that modern instant messaging platforms are how on-demand developer documentation has been (maybe poorly) implemented in the last decade.

In the modern era of software documentation, a new paradigm follows in the wake induced by the emergence of platforms like GitHub and StackOverflow. They fundamentally changed how developers (and users) communicate about software projects: Mailing lists and forums are declining in favor of multi-media instant messaging platforms, such as Gitter [50, 66, 112, 174, 175, 209, 210, 225], Slack [43, 144, 174, 210, 241], Discord [113, 141, 181, 182, 183, 244], and modern forums like GitHub Discussions [95, 143].

Recent years have seen a shift in what kinds of documentation are usually associated with software systems. In the era of collaborative development on GitHub, we have non-authoritative crowd-sourced documentation (*e.g.,* StackOverflow) that grows independently of the referenced system. More popular projects can generate more documentation, but this is ever more decoupled from source code, without clear strategies to address its maintainability [23, 153, 156]. The authoritativeness problem can be partially mitigated by forms of consensus (*e.g.,* answer voting on StackOverflow), but traceability and up-to-dateness are often overlooked. Some approaches have been proposed to reconnect external documentation sources and source code in the IDE [176], or to leverage, for example, social media in software maintenance [158]. Nevertheless, a fundamental understanding of the paradigm shift affecting modern software documentation is still missing. We need to evaluate the impact and dynamics of the subtle but constant drift induced by new communication and documentation platforms.

1.2 The Software Documentation Landscape

Software documentation sources are heterogenous and continuously evolving, subject to external and internal forces. To explain how these forces influence software documentation we propose a new metaphor: The *software documentation landscape*.

Software documentation is a fundamental asset of every system and its evolution needs to be understood as a single collective phenomenon. Documentation is like emerged landmasses on Earth; stakeholders form settlements on these landmasses and shape them according to their needs. If classical software documentation is the *Pangaea* of tectonic plates and continental drift theories [260] (*i.e.,* when all documentation is close together and to source code), modern software documentation is becoming a number of archipelagoes that should be mapped, explored, and linked back to the mainland (*i.e.,* source code).

As a first intuitive definition, the documentation landscape of a software system is the composition of all the possible sources of information able to support activities pertaining to the design, implementation, comprehension, maintenance, and evolution of the system, performed by different stakeholders of the software project, and in particular by developers. Making the landscape explicit and developing the tools to capture and analyze its evolution elicits a higher level of understanding of what software documentation is becoming.

The documentation landscape captures the value of documentation as a fundamental asset of a software system. The evolution of the landscape provides information about its history and current direction. Finally, from the analysis of multiple landscapes of different systems, trends emerge. Our work provides insights into opportunities, but also perils, for current and future accessibility of software documentation.

Despite being such a fundamental asset, documentation is often considered an afterthought by software developers [6]. Prone to being neglected by its own nature, sometimes, software documentation is also neglected by researchers.

We identify five open challenges affecting modern software documentation:

- Modern software documentation is heterogeneous. Multiple media coexist, competing for attention, sometimes presenting similar content in different forms. A taxonomy and a systematic mapping of documentation are missing.

- The migration towards instant messaging applications of developer communities is a "silent killer" of software documentation, with an unknown (and hard to foresee) impact on long-term reliability and retrievability of documentation.

- Documentation standards depend on the project's development methodology and stakeholder interests. A guidance framework for consistency is missing.

- Recent evolution of documentation is a complex phenomenon affecting multiple dimensions (*e.g.,* authoritativeness, persistence, retrievability). The interplay between different dimensions is unexplored.

- Last but not least, a comprehensive characterization of the landscape of modern software documentation is missing and current research tends to be conducted as in separate format- or application-dependent information silos. For example, most of the papers that we analyze in the state-of-the-art (Chapter 2) take into account a single form of documentation, or just few, comparing them in similar applications.

With our work, we aim to provide a holistic view of the nuances of software documentation. The different dimensions of the software documentation landscape constitute a defining and differentiating gradient for documentation sources. This gradient makes some sources more important than others during the evolution of the landscape. An overview of this phenomenon that encompasses the evolution of multiple documentation sources is currently missing in the literature. This leads us to the central thesis of our work.

1.3 Thesis

Documentation has always been about communication. Communicating intentions, designs, or functional behavior of software systems. This communication nature is pulling documentation closer to modern communication infrastructures, influencing and being influenced by characteristics of newer media. The natural tendency of developers to consider writing documentation an unpleasant endeavor [135, 235, 258] only adds to the problem by increasing the importance of collecting, persisting, and summarizing documentation, also when it is informal or unstructured. Mapping and reifying the documentation landscape will provide a framework to interpret, capture, and influence the complex phenomena affecting modern software documentation.
We formulate our thesis as follows:

Thesis Statement

"The software documentation landscape is the heterogeneous multitude of documentation sources useful for different stakeholders of a software project. It is a fundamental asset whose evolution needs to be harvested, reified, persisted, and summarized to feed back into the evolution of the practice.
By developing the landscape metaphor we provide a holistic interpretation of the phenomena affecting modern software documentation and the means to understand their implications."

— Marco Raglianti, 2025

1.4 Contributions

We highlight the main contributions of our research activity in support of our thesis statement. We use the following legend to indicate the type of each contribution:

- Insights, reflections, and threats.
- Publication in a peer reviewed conference.
- Empirical study.
- Taxonomy or similar classification of landscape entities.
- Published dataset (*e.g.,* replication package) to promote further studies.
- Software tools to extract, analyze, or visualize the landscape (whole or part).
- Co-supervised Master Thesis.

We link each contribution to the parts of this thesis (*e.g.,* chapters, sections, figures) where the relevant results are presented and discussed.

Our main contributions consist of:

State of the Art: Half a Century of Software Documentation in Software Engineering Chapter 2

We highlight key works in the history of software documentation, from the efforts in the '70s to scale documentation practices to a field tackling projects of ever growing complexity and unprecedented scales, to an analysis of current research on the use of modern communication platforms for software engineering. In Chapter 2, we present the relevant literature to contextualize the different parts of our work.

Outlining the Documentation Landscape Chapter 3

We analyze the evolution of the documentation landscape in a dataset of relevant open source projects. In Chapter 3, we present the approach and the insights emerging from the large scale analysis on the full dataset and the single case studies.

A Taxonomy of Documentation Source Types Figure 3.5

We devised a taxonomy of the landscape from the progressive refinement of the regular expressions used to automatically capture the sources of the software documentation landscape across the last two decades of its evolution.

Mapping the Documentation Landscape of Open Source Projects [196]
Student: Tommaso Rodolfo Masera
Advisor: Michele Lanza, Co-advisors: Csaba Nagy, Marco Raglianti.
Università della Svizzera italiana, Lugano, Switzerland, A/Y: 2022–2023.

An Automatic Mapping of the Documentation Landscape and its Main Dimensions Chapter 4

We analyze the documentation landscape from the point of view of its constituent dimensions. We consider form and contents, as well as instances of popular platforms that act as catalyzer to form de-facto standards for parts of the documentation landscape. We show a trend currently *pushing* software documentation towards synchronous communication channels, deemed more apt to leverage the faster but (dangerously) more volatile new media (*e.g.,* modern and media-rich instant messaging applications). In Chapter 4, we present insights that range from a broad overview encompassing the transition away from traditional documentation to a deep dive into the communities hosting modern software documentation.

On the Rise of Modern Software Documentation [184]
Marco Raglianti, Csaba Nagy, Roberto Minelli, Bin Lin, Michele Lanza.
In *Proceedings of the European Conference on Object Oriented Programming (ECOOP)*, no. 43, pages 1–24, 2023, Dagstuhl Publishing.

Dataset of Manually Annotated Documentation Sources Section 4.3.3

We published a dataset of 60 projects where we manually annotated the top-level README file with the documentation sources directly linked by developers. In Section 4.3.3 we present how the dataset has been obtained.

Dataset of Automatically Extracted Evolution of Communication Platforms Documentation Sources Section 4.3

We published a dataset of the evolution of communication platforms in 13k open source projects, where we automatically extracted the evolution of synchronous, asynchronous, and social communication platforms directly linked by developers. We include in the dataset information about the evolution of communication platforms like Discord, Gitter, and Slack in the last 10 years. In Section 4.3, we present how the dataset has been obtained with DwarvenMail, the scraper tool providing the link classification heuristics.

Analyzing 20 Years of Evolution of the UML Documentation Landscape Chapter 5

We show an example of how relevant artifacts in the UML landscape can shed light on trends about software documentation practices. Our analyses provide guidance for future research and development efforts to overcome current limitations of UML authoring tools and practices, also thanks to a holistic view of the landscape. In Chapter 5, we present our approach, the research questions, and their answers to (re-)assess the past and investigate the present and future of UML documentation.

Investigating the Past, Present, and Future of UML Documentation in Open Source Software [199]
Joseph Romeo, Marco Raglianti, Csaba Nagy, Michele Lanza.
In *Proceedings of the International Conference on Software Engineering (ICSE)*, in press, 2025, IEEE.

Dataset of Automatically Tagged UML Artifacts Sections 5.3–5.4.2

We published a large dataset of automatically tagged files in open source repositories comprising UML artifacts versioned in GitHub in the last two decades. In Sections 5.3–5.4.2 we describe our approach to the mining and annotation of the dataset.

On the Usage of UML Diagrams in Open Source Projects [197]
Student: Joseph Romeo
Advisor: Michele Lanza, Co-advisors: Csaba Nagy, Marco Raglianti.
Università della Svizzera italiana, Lugano, Switzerland, A/Y: 2022–2023.

In-Depth Analysis of the Discord Instant Messaging Application for Developers Communication — Chapters 6, 7, Section 8.1

Discord is one of the actually most popular instant messaging applications among developers of open source projects. Hosting huge communities of developers, it is a yet untapped gold mine of information about unstructured and spontaneous crowd-sourced documentation. With progressively deeper analyses, we highlight the relevance but also the issues of this new documentation source. In Chapters 6, 7, and 8 we present our approach to visualize the information present in Discord servers, and in particular in developers' conversations.

DiscOrDance — Section 8.1

In Section 8.1, we present the tool we use in multiple studies involving Discord servers used by developers for communication.

Visualizing Discord Servers [181]
Marco Raglianti, Roberto Minelli, Csaba Nagy, Michele Lanza.
In *Proceedings of the Working Conference on Software Visualization (VISSOFT)*, pages 150–154, 2021. IEEE.

DiscOrDance: Visualizing Software Developers Communities on Discord [182]
Marco Raglianti, Csaba Nagy, Roberto Minelli, Michele Lanza.
In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 474–478, 2022, IEEE.

Using Discord Conversations as Program Comprehension Aid [183]
Marco Raglianti, Csaba Nagy, Roberto Minelli, Michele Lanza.
In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 597–601, 2022, ACM.

Capturing and Understanding the Drift Between Design, Implementation, and Documentation — Section 8.2

Thanks to the analysis presented in Section 5.1, we could leverage an ongoing paradigm shift in the UML documentation landscape. We propose a novel approach to link UML artifacts (*e.g.,* diagrams) and source code entities (*e.g.,* classes, methods), leveraging emerging UML human-readable textual formats. Through DRIFTER we capture and visualize the drift between design, documentation, and implementation and its evolution.

Capturing and Understanding the Drift Between Design, Implementation, and Documentation [198]
Joseph Romeo, Marco Raglianti, Csaba Nagy, Michele Lanza.
In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 382–386, 2024, ACM.

Visualization of the Evolution of the Software Documentation Landscape — Various Figures

Preliminary evolution of the aggregate landscape → Figures 3.19, 3.20, 3.21.
Partial evolution of the instant messaging applications landscape → Figure 4.8.
Evolution of the UML documentation landscape → Figures 5.2 and 5.3.

1.5 Structure of the Document

We introduced the context of the thesis, the importance of documentation as critical asset for software systems, problems with its current state, and how mapping and reifying the documentation landscape could shed light on the complex phenomena affecting modern software documentation.

In Chapter 2, we present role and characteristics of software documentation, from the early history of software engineering to the many forms of modern software documentation. We also present relevant works on communication platforms and instant messaging applications studied as documentation sources for software engineering practices.

In Chapter 3, we present a preliminary study on how to map the documentation landscape. We give a definition and an initial taxonomy, a mapping of the landscape in more than 10 thousands relevant open source projects, and its evolution over two decades. We show how we visualize the evolution of the landscape as a whole and in three case study projects.

In Chapter 4, we analyze how modern software documentation is shifting towards fast and volatile communication channels (*e.g.,* instant messaging applications). Starting from a broad overview we highlight coexistence and competition patterns, narrowing down the evolution to the communication platforms and the resulting communities.

In Chapter 5, we analyze the past, present, and future of the UML documentation source. We study old and new formats of UML artifacts. Currently popular ones hint at a resurgence of UML as documentation means, with new opportunities and new issues to be addressed.

In Chapter 6, we present, as a case study, an instant messaging source that has become popular among software developers: Discord. We detail our approach to model, mine, and analyze developer communications (including source code snippets) in a Discord server, a relatively unexplored source, yet more relevant than its more investigated alternatives (*e.g.,* Slack, Gitter).

In Chapter 7 we present an approach to refine our model of a Discord conversation as a first class concept. We show ways to visualize its contents to support a higher level comprehension of developer communications involving source code. We also present the problem of conversation disentanglement and CoDi, our first step towards extensible, configuration-less disentanglement of interleaving messages in Discord.

In Chapter 8, we present two of the tools we leveraged in our analyses in more detail: DiscOrDance and Drifter. We introduce the architecture and the user interface of DiscOrDance, showing its extensibility via view specifications. With Drifter, we present our approach to capture the drift between design, implementation, and documentation artifacts (DID drift).

In Chapter 9, we summarize our contributions and outline the short term future work and our long term vision for the documentation landscape. Before a few closing words, we give a glimpse of the uncharted territory consisting of large language models for software documentation.

The Appendices contain technical details of RagnaDok, the tool we used to mine and analyze the documentation sources in our preliminary study (Chapter 3).

In Appendix A, we show the architecture of RagnaDok, how it retrieves readme files, and how it leverages the *git* log to reconstruct readme histories from commits.

In Appendix B, we list RagnaDok's detection methods to automatically identify documentation sources for each leaf category in our taxonomy.

2

Half a Century of Software Documentation

We analyze the role and characteristics of software documentation since the birth of software engineering, up to its current state in global software development. Then, we present works related to important qualities for useful software documentation. Finally, we conclude with related research about instant messaging applications and how they changed developers' communication, becoming a form of on-demand software documentation for developers communities.

2.1 Early History of Software Engineering

Computer software is comprised of programs and documentation since its inception [231]. When software development on a large scale started to be recognized as an engineering discipline [163], generation of software documentation was already an established practice (and a burden) with specific aims, advantages, and criticalities [163].

Naur identified the importance of documentation, especially during production, as an essential part of software aimed at the human reader [163]. According to Selig [219], "*documentation* [was] *becoming essential, not just to save money but to prevent chaos.*" Particular emphasis was put on the need of different documentation for different readers. In Opler's concerns on the lack of testing for documentation accuracy and clarity [167] there was a distinction between *user* and *operator manuals*, two prevalent forms of documentation at the time. In his working experience at IBM, Opler also identified the need for documentation to be up to date and how its growth could be used as a proxy measure of project velocity [163].

Documentation was considered necessary to support software systems. Tausworthe characterized documentation needed for long-life-cycle programming systems according to quality and level of detail [248]. Usefulness of documentation was constantly challenged (*e.g.,* flowcharts [186, 229], program design languages [186]) but evidences emerged showing the impact of documentation format on the performance of programmers in software related tasks [223].

In the early history of software engineering, besides physical manuals, there was a form of documentation tightly coupled with source code that is still extensively studied nowadays [115, 233]: *Code comments.* In 1956, the specification of *Fortran I* already included comment statements as part of the official language [19]. The importance of comments was recognized and agreed upon. According to style and programming guides of the time, code comments should have been accurate and useful [123, 133]. Code comment analysis dates back to the origins of software engineering as well. In his analysis of PL/I programs, Elshoff found a relationship between (the lack of) code comments and code readability and maintainability [67]. Brooks argued that code comments are a bridge between knowledge domains in program comprehension [36].

2.2 Software Crisis, Processes, and Mental Models

In the words of Dijkstra [60], "*the turning point was the Conference on Software Engineering in Garmisch [163], October 1968, a conference that created a sensation as there occurred the first open admission of the software crisis.*" This awareness would shape the field of software engineering for the following decades. Endres named it the era of "mastering the process" [70], whose focus was to identify and address common causes of failure for software development. In phase-based development (*e.g.,* the waterfall model [202]), code inspections [74] were the perfect example of practices introduced to mitigate the high risk of failure of an ever growing industry, with projects constantly increasing in size and complexity [163, 230]. Although the focus was on the process, Royce deemed design documentation an important upfront cost to be paid for advantages downstream in the testing, operations, and redesign phases [202].

While recognizing the engineering spirit of software development, Parnas and Madey went towards a formal definition of documentation content [172]. Their aim was to represent properties of computer systems as a set of mathematical relations, to reduce ambiguity and the need for (costly) redesign at system integration time. *System Requirements Document*, *System Design Document*, *Software Behavior Specification*, and *Software Module Guide* were a few of the technical documents formalized by Parnas and Madey but, in their own words, "*many of the ideas presented in* [the] *paper* [were] *old by computer science standards.*"

At the same time, steady growth in size and complexity of software made program comprehension and maintenance tasks more relevant [33, 188, 204, 205]. Mental models of software started with Brooks twenty years before [36] evolved into computer supported generation of documentation [236], while software visualization techniques were employed to aid program comprehension [75, 111, 126, 128, 148, 237, 245]. Documentation, also the one generated on-demand by visualization tools, supported the formation of mental models to understand software [238].

2.2.1 UML and Architecture Erosion

Child of the *process era*, the Unified Modeling Language (UML) is a visual modeling language with a formal specification, standardized in 1997 by the Object Modeling Group. Touted as the de facto standard for modeling software systems from structural and behavioral points of view, UML allows developers to visually describe the system, what it does, and how. Specialized software is used to create UML diagrams [107], with an emphasis on their syntactic correctness.

Software engineering courses in university computer science curricula teach UML to students as a tool to reason about software, its design, and its documentation [71]. The relevance of UML in teaching is not matched by its usage in practice, though: In a survey with 80 software architects, Lange *et al.* found that UML in industry only loosely adheres to the official specification [130]. Nevertheless, UML is a valid instrument to teach topics related to software architecture. Rukmono and Chaudron found that peer feedback on UML diagrams can improve students' learning of software design [206].

High-quality documentation increases the success chances of a software project [64, 135, 255]. This is especially true for UML diagrams and documentation based on visual modeling. Software design diagrams promote active discussion in homogeneous and cross-functional teams [118]. Developers achieve better functional correctness when making changes with accurate and up-to-date design diagrams [11, 64].

Although most results agree on the usefulness of UML, some exceptions exist. Scanniello *et al.* found that UML models produced in the requirements analysis process influence neither the comprehensibility nor the modifiability of source code [213].

Gravino *et al.* found that the time to accomplish comprehension tasks for less experienced participants is negatively affected when UML design models are provided in addition to source code [90]. There is a necessary trade-off between correctness, quality, and time to complete software maintenance tasks when leveraging UML diagrams [64].

UML drawing software tools embrace the visual nature of the language, providing drawing capabilities similar to whiteboard sketches, mixing pen and paper with digital means [22, 46, 53]. Bergström *et al.* investigated the automated assessment of UML class diagram layouts via machine learning, since element spacing and rectangle orthogonality play a fundamental role in the perceived quality of a UML class diagram [29]. Human-readable text-based UML takes away control from the diagram's creator, for example, on how elements are laid out, their spacing, and where connecting lines are drawn. This impairs what Rukmono and Chaudron found to be key aspects of understandability and usability of UML diagrams [206].

Badreddin *et al.* investigated the underutilization of models in open source software (OSS) and a potential language-based solution: Umple [20]. Robles *et al.* and Hebig *et al.* already investigated UML models in the lifespan of GitHub projects by focusing on images and a limited subset of non-image file extensions (.uml and .xmi) [98, 195]. While the classifier proposed by Ho-Quang *et al.* [101] has been utilized to identify class diagrams, manual analysis was performed on the remaining images to discriminate other types of diagrams.

Previous studies used surveys with developers and software practitioners to investigate their perception and usage of software modeling [78]. Ho-Quang *et al.* analyzed the role of modeling in OSS development [102]. Analyzing projects containing UML files, they used a method similar to ours to retrieve UML files for project selection, but our study aims to address different research questions and to provide empirical evidence based on the analysis of a large dataset of UML artifacts spanning 20 years. Besides updating the analysis of UML usage in OSS with the trends in the last 6 years, the phenomenon emerging between 2017 and 2022 present new and distinctive characteristics that warrant specific attention.

Our work focuses on the evolution of text-based representations of UML as a more reliably parsable and easily manageable format and the Umple format is one of those we analyzed.

The resurgence of UML with human-readable text-based formats, one of the main findings of the study presented in Chapter 5, is indeed an emerging phenomenon that prompts for better approaches and tools. For example, we need to address layout issues, without giving back the ease of use and maintainability gained by making the graphical representation almost a side-effect.

2.2.2 UML and Software Documentation

High-quality documentation increases the success chances of software projects [64, 135, 255]. UML diagrams, although with some notable exceptions [90, 213], support developers by promoting active discussion in teams [118] and by making them achieve better functional correctness when changes can leverage accurate and up-to-date diagrams [11, 64].

Model-Driven Reverse Engineering (MDRE) [185] reconstructs model descriptions of existing systems, for example, for documentation, migration, and evolution of legacy systems (see the approach of Favre [76] or tools like MoDisco, from Brunelière *et al.* [37]). Sabir *et al.* investigated how to generate UML diagrams from existing Java source code with MDRE techniques [208].

The natural divergence between architecture and implementation has been extensively studied. Previous studies on architecture erosion [16, 55, 137, 138, 139], degradation [100], and consistency [8, 200] assume a design-first approach, where the focus is on the effect of development on the intended architecture of the system.

In our UML artifact analysis (Chapter 8), we focus on human-readable text-based UML formats because of their increasing popularity in teaching and practice (for example, see Carruthers *et al.* [40] and Jasser *et al.* [110]). We show how UML can also be used as *a posteriori* documentation and we consider the role of design documents fulfilled when they become documentation for the comprehension of the implemented system.

MDRE focuses on creating new diagrams from code, our approach leverages existing UML artifacts to analyze the status and the evolution of design, implementation, and documentation of a software project, especially in the early phases of projects adopting a design-first approach.

To the best of our knowledge, our work is the first to consider a complete development iteration, from design, through implementation, to documentation, highlighting the evolution of the gap that naturally tends to affect artifacts produced in the three phases.

2.3 The Agile Movement

Multiple independent experiments of counterculture against process-oriented software engineering acquired momentum in the late 1990s. These *agile methodologies* (*e.g.,* Extreme Programming [27], SCRUM [217]) coalesced in an agreed upon manifesto with a shared vision [82]. The *agile movement* aimed at removing all of the burdens deemed unnecessary to release early and release often [187], valuing "working software over comprehensive documentation" [82]. There was no place for formal, limiting, and time-consuming documentation, soon becoming obsolete with the fast paced development methodologies in the new millennium.

In Extreme Programming, documentation was replaced by simple design, common metaphors, collaboration among users and programmers, and frequent verbal communication [27]. System specification took the form of automated acceptance tests [27]. As a result, developers felt entitled to neglect documentation. At least until a decade later, when a renewed research interest hinted at its vital role for successful development and maintenance of large systems, even when agile methodologies were involved [28, 93, 94, 103, 179, 203, 218], without significant differences from structured development [72] and with effects that undermined some of the agile predicaments about software documentation [235].

Shaw identified a shift, since the 1960s and across four decades, from writing small programs to analyzing and reasoning about large distributed systems [221]. Small co-located groups dominating software development in the 1960s and early 1970s could rely on face-to-face communication [239]. This principle was brought forward and reinforced in the value of face-to-face communication in the *Agile Manifesto* [82], but the landscape was already changing under accumulating external pressure (*i.e.,* internet and social networks).

2.4 The Modern Age of Software Documentation

A plethora of new documentation sources emerged in the last two decades. It started with widespread access to the World Wide Web and the push towards the read-write *Web 2.0*. Leuf and Cunningham defined the Wiki Way [136], a form of participated documentation and an online style of communication that reduces the entry barrier for contributors. Crowd-sourced documentation became a reality [160, 173, 211]. Users and other stakeholders could be involved in the creation of documentation. This (r)evolution [239] reached its apex with global knowledge exchange and Q&A platforms like StackOverflow, extensively studied in the last ten years and an unexhausted-yet mine of software documentation insights [21, 23, 31, 44, 47, 85, 145, 153, 156, 173, 176, 211, 264].

A few examples of modern software documentation are: Developers' blog posts [170], software engineers' microblogging on Twitter [251], rich-media instant messaging [43, 66, 144, 175, 183, 225, 244], news aggregators [10], technology tutorials [12, 13], and feature-rich forums [95, 143].

By mining these sources it is possible to complement and fix classical documentation [39, 176], improve software maintenance (*e.g.,* bug prioritization and triaging) [158, 211], automatically generate documentation for comprehension [4], or discuss technical aspects and share new software releases [220]. According to Constantino *et al.*, globally distributed software development, especially in open source software, often includes documentation tasks as a gateway to start knowing a project and to collaborate on it [48].

Modern software documentation is considered a critical asset for developers [5]. It has been studied extensively with respect to its quality and usefulness [6, 26, 45, 54, 79, 83, 192, 231, 265]. Documentation can suffer from a number of issues [6]. In particular, outdatedness impacts its usefulness [191, 192]. Documentation is regarded as an integral part of the software development process [171] with various uses in the lifecycle of software [231] (*e.g.,* as a communication medium for stakeholders, information repository for maintainers, and guide for software users).

Nevertheless, fifty-years-old problems [67, 123, 133] still affect even the documentation form closest to source code. For example, Steinbeck and Koschke analyzed Javadoc comment violations in 163 OS projects [233]. They concluded that about half of the elements do not have a Javadoc and when a violation is present it can live up to two years before being fixed. Regardless of the imminence of Javadoc comments, a de-facto standard for Java applications, and their proximity to source code, Javadoc low update rate is impairing its usefulness for software maintenance.

Continuous software engineering practices (*e.g.,* containerization, script-based DevOps) are expected to reduce the need for extensive deployment documentation [125]. Nevertheless, the gap between theory and practice of software documentation still persists, only partially justified by economic considerations about the cost-benefit ratio of non-executable artifacts [193].

2.4.1 Attributes and Usefulness

Software documentation comes in different forms and for different purposes. With such wide spectrum, it is natural that practitioners find some of the attributes of documentation more important than others with respect to clarity, usefulness, or in relationship to the task that the specific type of documentation should support.

Forward and Lethbridge surveyed software professionals about possible attributes of software documentation [79]. According to participants, the five most important contributions to effectiveness are (in decreasing order of importance): content, up-to-datedness, availability, use of examples, and organization (*e.g.,* sections, subsections).

Documentation is useful for design, development, testing, maintenance, and operations [5, 26, 45, 258]. Chen and Huang show how documentation quality is one of the five dimensions that affect software maintainability [45]. Documentation problems are a key aspect determining maintainability level of software. Four documentation issues out of five are in the top-ten severity problem factors. Dagenais and Robillard investigated documentation production modes (initial effort, incremental changes, and bursts) and their relationships with decision points in OSS development [54]. They analyzed how Wikis are selected as starting documentation infrastructure to encourage crowd-sourced documentation (*i.e.,* expect users to also create and update documentation). Wikis are chosen initially to lower the entry barrier for contributors, but one out of three projects moves to more controlled infrastructures to improve documentation reliability.

Different types of documentation are better suited for different tasks [5]. Garousi *et al.* analyzed the use of technical software documentation for software development and maintenance [83].

In an industrial case study, they found that technical documentation is consulted least frequently for maintenance and, together with source code, most frequently for development. They also confirmed that up-to-dateness and accuracy of information have the highest impact on the usefulness of technical documentation [83]. They did not find any statistically significant difference between the usage of various documentation types in development or maintenance activities.

Kajko-Mattsson presents requirements for product and process documentation useful for corrective maintenance [120]. The importance of consistency, correctness, completeness, and traceability is recognized, but practices to ensure these properties are severely lacking. For a complete overview, Zhi *et al.* present a systematic literature review of costs, benefits, and quality of software development documentation [265]. Ding *et al.* provide a review of knowledge-based approaches to software documentation [61].

2.4.2 Communication and Instant Messaging

Communication channels, especially those tightly coupled with collaborative development platforms (*e.g.,* GitHub), are fundamental for successful software development.

Hoegl *et al.* [104] and Lindsjørn *et al.* [147] found communication to be an integral component of teamwork quality. Tantisuwankul *et al.* analyzed the communication channels of GitHub projects [247]. Studying 70k library projects in 7 ecosystems, they identified 13 communication channels as "a form of knowledge transfer or sharing" (*e.g.,* licenses, change logs). They found that GitHub projects adopt multiple channels, which change over time, to capture new knowledge and update the existing one.

Storey *et al.* [240] conducted a large-scale survey with 1,449 GitHub users to understand the communication channels developers find essential to their work. On average, developers indicated they use 11.7 channels across all their activities (*e.g.,* email, chat, microblogging, Q&A websites). They concluded that "communication channels shape and challenge the participatory culture in software development." Hata *et al.* [95] studied early adopters of GitHub Discussions, finding that developers considered them useful and important. Lima *et al.* [142] used NLP to detect related discussions of OSS communities in GitHub Discussions.

Treude and Storey [254] interviewed users of a community portal, finding that clients, developers, and end-users are involved in the process of externalizing developer knowledge.

Nugroho *et al.* [165] studied how Eclipse developers utilize project forums, concluding that forums are essential platforms for linking various resources in the Eclipse ecosystem, besides representing an important source of expert knowledge.

Modern social media are becoming another information source for development activities. Mezouar *et al.* [158] studied how tweets can improve the bug fixing process. They observed how issues for Firefox and Chrome are usually reported earlier through Twitter than on tracking systems. This can potentially decrease the lifespan of a bug. Guzman *et al.* [91] analyzed the usage characteristics, content, and automatic classification of tweets about software applications. They found that tweets contain useful information for software companies but stressed the need for automatic filtering of irrelevant information.

Instant messaging platforms, from Internet Relay Chat (IRC) to Discord, went from simple text messages to rich multimedia support with integrated DevOps workflows (*i.e.,* Slack integrations). Yu *et al.* [263] learned how real-time (*i.e.,* IRC) and asynchronous (*i.e.,* mailing lists) communications were used and balanced across the GNOME GTK+ project. Shihab *et al.* [226] analyzed IRC meeting logs and found how developers contributed through meeting channels.

Lin *et al.* [144] argued that Slack was playing an increasingly significant role in developers activities and coordination, sometimes replacing emails, with various benefits over mailing lists.

Developers use Slack for team-wide purposes (*e.g.,* communicating with teammates, file and code sharing), community support (*e.g.,* special interest groups), and personal benefits (*e.g.,* networking, social activities). Lin *et al.* also observed that developers commonly use bots to support their work [144]. Chatterjee *et al.* [43] analyzed the conversations of developers from five Slack programming communities and developers' StackOverflow posts. They found useful information prevalent, including API mentions and code snippets with descriptions in both sources.

Alkadhi *et al.* [9] examined "rationale" elements (*i.e.,* discussed issues, alternatives, pro-/con-arguments, decisions) in Atlassian HipChat messages of three software development teams. They found frequent, valuable discussions with elements of rationale. However, they also emphasized the need for automated tools due to the high volume of chat messages.

Shi *et al.* [225] conducted an empirical study on Gitter chats. They manually analyzed 749 dialogs and performed an automated analysis of over 173k dialogs of OSS communities. *Interestingly*, developers tend to converse more on Wednesdays and Thursdays. They also found interaction patterns among conversations and noticed that developers tend to discuss topics such as API usage and errors. They argue in favor of a better mining and utilization of knowledge embedded in the massive chat history of OSS communities.

Hata *et al.* analyzed links in source code comments [96] in a large-scale study (∼10 million links) extracted from files of the main language of the project.

Ebert *et al.* [65] conducted an empirical study to understand which communication channels are used in GitHub projects and how they are presented to the audience, finding that the most common were chats, mail-related, social media, and GitHub channels. Käfer *et al.* [119] analyzed GitHub communication channels, finding that "Mailing lists are being replaced by modern enterprise chat systems in OSS development."

Each of the previously discussed studies focuses on a specific part of the documentation landscape, recognizing the importance of the sources for knowledge management and documentation. What is still missing is a higher level understanding of the phenomenon that shifts the relative importance of these sources over time, intra- and inter-project.

Communication is one of the main motivations to produce software documentation [79, 231] and it is related to bug introducing changes [1, 30]. In particular, Instant Messaging (IM) applications are a pillar of developer communication [49], especially in global software development [177, 178, 226, 249]. They are almost supplanting more traditional media [119] (*e.g.,* emails, mailing lists). IM stands out for its speed, availability, and richness of communication, sometimes providing a parallel "shadow system", outside of an organization's toolset. Developers resort to use it to get things done (*e.g.,* Lin *et al.* [144]). With multiple communication channels available, though, also comes the need to have communication policies to avoid adverse effects (*e.g.,* partial, excessive, unreachable information) [48].

IM applications can be born in various contexts (*e.g.,* virtual workspaces, gaming) and become popular among developers, or they can be explicitly created to support developer communities. Slack and Discord are examples of the former case, while Gitter is an example of the latter, tightly coupled with the GitHub collaborative development platform. We present important studies on the most relevant IM applications and their use in project development and management.

In the work presented in Chapters 3 and 4, we focus on readme file links, which are independent of the project language. Our work broadens the scope beyond communication channels and adds details needed to identify the current status, understand how it has evolved, and obtain meaningful insights on why this is happening.

2.4.3 Instant Messaging Platforms for Software Development

Among the many possible instant messaging applications, there are some that caught the attention of software developers and, consequently, of software engineering researchers. From the oldest text-only chats (*e.g.,* IRC) to modern rich-media applications, supporting the integration of workflows (*e.g.,* Slack), there are a few works which investigated the use of such platforms among developers communities. We present the works on four of the more widespread platforms.

Internet Relay Chat (IRC)

Text-based IM system supporting group channels, direct messages, and file sharing. Mutton proposes an approach to infer and visualize social networks in IRC channels [161]. Shihab *et al.* analyzed the use of IRC to coordinate global software development in the GNOME GTK+ project [226, 227]. Elsner and Charniak proposed an approach to disentangle conversations in IRC [69]. Their work highlights the importance to aggregate IM messages in higher-order structures (*i.e.,* conversations, threads), subsequently leveraged also in the analysis of other applications (*e.g.,* Slack [42], Discord [183, 244]). Chowdhury and Hindle evaluate two machine learning algorithms to detect off-topic discussions in IRC channels related to programming [47]. In general, although still used by some communities, text-based IRC has been almost completely supplanted by more recent IM applications.

Gitter

IM system targeted at users of GitHub and GitLab, directly connects chat rooms to repositories by using shared user access control mechanisms and allowing easy file and issue linking during conversations (*e.g.,* by adding *#issue_number* in the message).

Parra *et al.* provided a curated dataset of 10,000 manually labelled Gitter messages [175]. Parra *et al.* showed that Gitter is an alternative to other platforms (*e.g.,* StackOverflow, forums) to fulfill active development tasks, to ask for help with development challenges, and to discuss new technologies [174]. Ehsan *et al.* focused on automatic reconstruction of discussion threads in Gitter [66]. Thread-based constructs provide contextual information that single messages lack. Shi *et al.* investigated when and how developers interact, what community structures look like, and which topics are discussed on Gitter [225]. API usages and errors are the most common topics, confirming the hypothesis that useful (on-demand [193]) documentation is created and consumed through IM. Parra *et al.* compared applicability of the categories identified by Lin *et al.* [144] for Slack to developer messages exchanged on Gitter [174]. Significant discrepancies emerged between self-reported and actual usage purposes. The qualitative study reported by Parra *et al.* also evaluated transferability of research results between IM applications, as previously done for IRC, Slack, and Discord [42, 183, 244]. Jiang *et al.* explored cross-platform contributors between Gitter and GitHub [112]. Sajadi *et al.* studied emotions in developers interactions on Gitter [210].

Slack

Rich-media virtual workspaces for IM in professional organizations support integration with other applications. Lin *et al.* analyze how developers use Slack for general communication and collaboration with teammates, especially in remote teams [144]. The study highlights extensive use of Slack integrations for development, deployment, and customer support. Chatterjee *et al.* created a dataset (and a curated ground truth) of disentangled Slack conversations to support development of software engineering tools [42, 43].

Crowd-knowledge construction and participatory approaches [264] characterize the efforts to answer challenging questions. API mentions and opinions on best practices and tools are more present in Slack than in StackOverflow posts [43] (*i.e.,* where they are explicitly forbidden).

Stray and Moe showed how employees in global projects spend slightly more than 16 hours per week between scheduled and unscheduled meetings in Slack [241]. Stray *et al.* analyzed communications over Slack of an agile virtual team of 30 people for more than two years [242]. With a mix of direct and group messages, the two most frequently found statements are about problem-focused communication and technical information (48% and 20% respectively).

Discord

Discord is the newest platform to be widely adopted by various developer communities [184]. Subash *et al.* apply approaches previously validated on Slack and IRC to provide a curated dataset of automatically disentangled Discord chat conversations for software engineering research [244]. Lill *et al.* analyzed reuse of previous conversations and posts in developers answers on Discord [141]. Discord still remains a relatively untapped source to mine for knowledge about software engineering practices and developer interactions.

For these reasons, our work on instant messaging documentation sources focused on the Discord platform. We developed DiscOrDance, a tool to extract and visualize the full history of a Discord server [182], including code snippets in syntax highlighted code blocks. We confirmed that source code is actually shared on Discord [181]. Coarse-grained structures, like conversations, can improve comprehension of topics and interaction patterns between developers [183].

2.4.4 Developer Communities

Numerous works have highlighted the importance of communication channels and platforms for a more comprehensive understanding of software systems and their developer communities.

Abreu and Premraj studied how software quality relates to communication frequency and the number of bugs injected into the software [1]. They showed that communication levels of key developers in the project do no correlate with the number of injected bugs, and that this number has a positive correlation with communication frequency.

Foundjem and Adams studied developers communications for coordination and synchronization in software ecosystems [80]. With a qualitative study on IRC meeting logs and Etherpad documents containing the release plan's status of the OpenStack ecosystem, they identified, catalogued, and documented synchronization activities in the software release cycle.

German *et al.* investigated ecosystems and their evolution with the GNU R case study [86]. In particular, they analyzed the communities that form around core and user-contributed packages. Communities form faster around core packages than around user-contributed ones. Moreover, developer communities tend to stagnate, while user communities grow unbounded.

Mutton studied social networks in IRC [161]. While his work focused on properties of generic networks through graph algorithms and metrics, his case study on the #java IRC channel showed features of a typical developers community (*e.g.,* high centrality of some elements, high closeness of the graph, identifiable disconnected sub-graphs).

Shihab *et al.* studied the increase in volume of conversations and number of participants of online meetings on IRC for the GTK+ and Evolution OSS projects [226, 227]. They identified dominant groups in both projects, with their contributions following, on average, a Pareto distribution (20% of participants contribute 80% of the messages), and stable contribution patterns.

Costa Silva proposed to analyze developer communication as a potentially suitable source of SE knowledge and how to present the resulting information in a useful way for developers [51].

Poo-Caamaño *et al.* studied communication and coordination in free and open source software (FOSS) ecosystems [177, 178]. They catalog communication channels and categorize high level communication and coordination activities. For the GNOME project, they identify three mailing lists, an IRC channel, Bugzilla, developer blogs, a wiki, and in-presence meetings (*e.g.,* conferences, hackfests) as communication channels with information of different types (*e.g.,* requests for comments, proposals and discussions, announcements) about the project.

Developer Communities Visualization

The software visualization community has neglected, so far, studies on developer communities, with some notable exceptions that we discuss.

Stephany *et al.* [234] proposed a visualization of software developer communities that depicts mailing lists and source code repositories of three open source projects. The authors highlight the underlying social structure and communication patterns between developers within each project.

Neu *et al.* presented a contributor-centric visualization in the form of "Developer Activity Diagrams" [164]. They mined *git* repositories to extract contributors' daily activities in terms of *git commits* at different scales (*e.g.,* project, ecosystem).

Goeminne and Mens [87] also mined and studied source code, mailing lists, and bug trackers. Their integration of different data sources to feed the visualization layer is another step in confirming the importance of communication between developers and between developers and users. Issue trackers, code repositories, wikis, and analytics platforms are possible knowledge sources considered for visualization by Johanssen *et al.* [116].

Stephany *et al.* visually analyzed mailing lists and source code repositories of medium to large developer communities with MAISPION [234]. Mutton presented graph-based visualizations of networks of users and their synchronous communications in IRC channels [161].

Campanella and Lanza proposed a visualization of developers activities in version control systems [38]. While emphasizing the disambiguation of multiple developer identities, their visualization helps in clustering groups of developers with similar activity patterns.

2.5 Summary

Documentation has been a constant companion of software since the birth of the programming discipline. The importance of documentation grew with the growth in size and complexity of software systems. Documentation encompasses a broad spectrum of contents, media, practices, uses, and stakeholders. Modern communication infrastructures emphasize even more the communication nature of knowledge management when documenting software, with a consequently faster, decentralized, but also more unpredictable and volatile form of documentation.

Evolution of practices and technical aspects of communication infrastructures play a crucial role in the phenomena affecting modern software documentation that we aim to capture and understand. This advocates for a systematic mapping of the software documentation landscape, of its current form, and its evolution.

To effectively investigate the possible documentation sources, and in particular the fastest and most volatile ones, we need to devise new approaches and new tools. To understand how documentation is evolving, we need to understand the rise in popularity of instant messaging applications as a form of documentation, a venue often referenced as a documentation source by developers themselves,[1] sometimes even as *the* primary documentation source, due to its low barrier of entry and highly dynamic nature.

[1] *"Ask for it in Discord!"*

Part II

Mapping the Documentation Landscape

An Overview of Sources and Their Evolution

3

Outlining the Documentation Landscape

We present our approach to gather and analyze data regarding the documentation sources that constitute the documentation landscape. We begin by defining what the documentation landscape is, and we explain how we extract the data necessary to represent the landscape in two main steps. Firstly, we analyze how README files evolve over time in a project. Secondly, we explore the data from the README to identify potential documentation sources in their content. The extracted sources are used to map the landscape of the project. We analyze and visualize the evolution of the software documentation landscape as a whole, as resulting from an empirical study across a large dataset of OSS projects, and more in depth for three case studies.

3.1 Documentation Landscape

The documentation landscape of a project is the composition of the content of all the documentation sources that are related to a software project. These sources can contain information about the system, for example its behavior in different scenarios (*e.g.,* a UML use case diagram), or about its usage, for example document installation instructions of third party libraries necessary for the system to be correctly deployed. The landscape exists implicitly and is navigable by developers who need to access some specific information for comprehension, maintenance, and evolution tasks. This implicit nature though, is the major obstacle to the effective retrieval of the required information. Sometimes the information is present but we even ignore the presence of the source, making retrieval virtually impossible.

This seems an unlikely scenario in the modern age of powerful search engines (*e.g.,* Google), domain specific knowledge bases (*e.g.,* StackOverflow), and formatted documentation automatically generated from source code (*e.g.,* Javadoc). However, a simple counter-example, that will be further substantiated in Chapter 8, is that Discord conversations are currently not indexed by search engines, and the popular communication application offers only limited functionalities to search across messages in its database [181, 182, 183].

Moving back to a higher level overview of the documentation landscape, to better understand its breadth and scope, we can compare it to a garden around a house (Figure 3.1). The house represents the system of interest, with its components. Surrounding the house there is the garden, whether properly maintained, left as a spontaneous wilderness, or simply empty. Each patch of flowers, flowerbed, and plant in the garden is a primary documentation source, maintained by the project owners (the same developers and maintainers of the system's codebase). Flower patches of similar flowers would then form a category of similar sources. Then, there is the public green surrounding the house, an intermediate region of crowd-sourced software documentation sources.

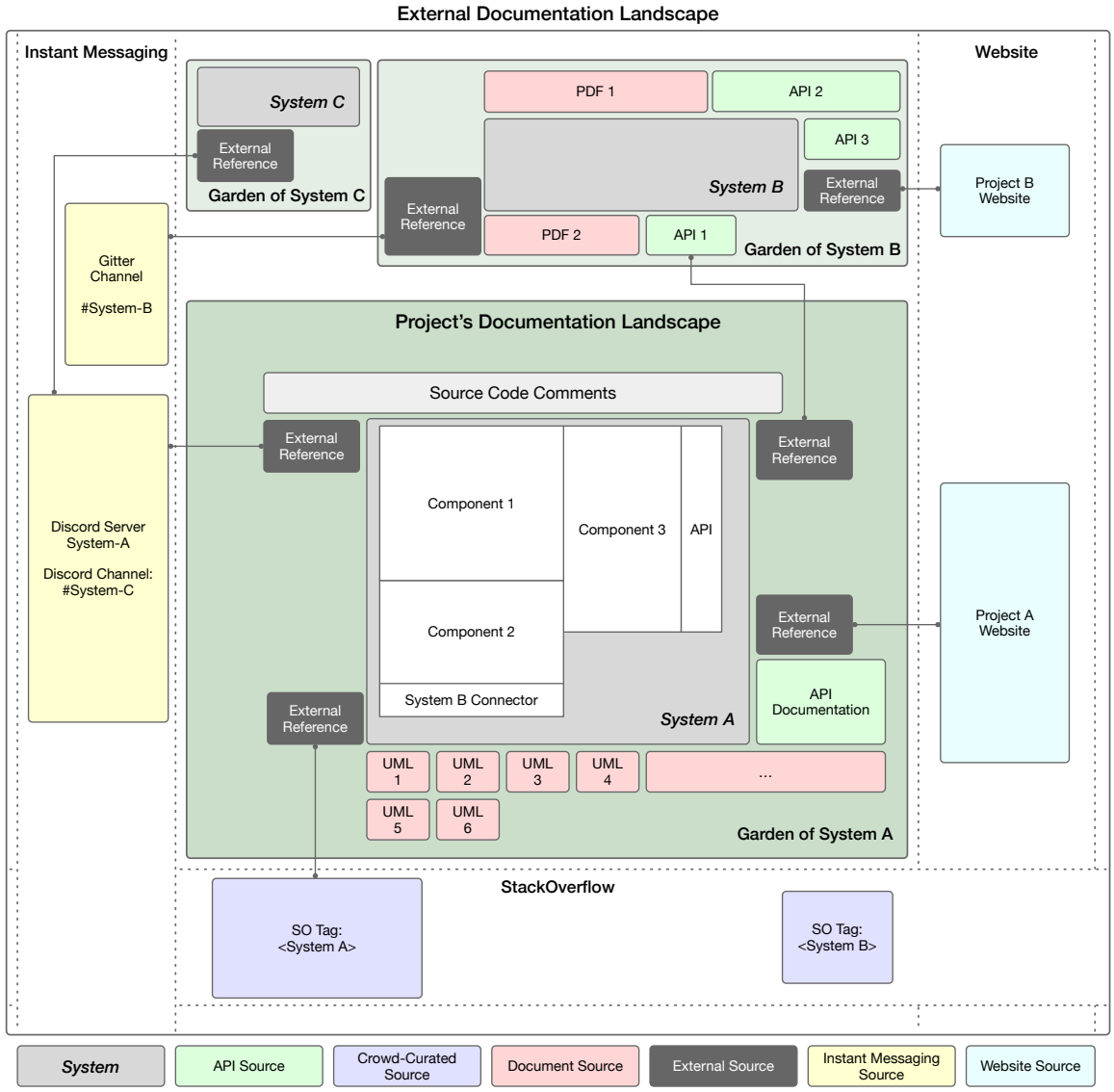


Figure 3.1: Software documentation landscape and the garden metaphor.

Maintained by different stakeholders, this part of the landscape is not under the direct control of the house owners, although they can influence it. This contributes nonetheless to the perceived value of the house and can be a useful asset, to perform activities useful for the house owners but for which the garden is not sufficient. Finally, there are the other houses, each with their own garden, close to the same or a different public park, but maybe with their favorite spot in a different corner under a different tree: The same crowd-sourced documentation source (*e.g.*, StackOverflow) hosts documentation for different projects, often attempting to classify it by different means (*e.g.*, tags).

The size and importance of the different components of the landscape may vary significantly, overall and with respect to one another. It is fundamental to have an accurate overview of the sources and their contents to address the documentation needs of a project.

3.2 A Preliminary Study of the Landscape and its Evolution

Our initial entry point to map the documentation landscape is a dataset acquired via the GitHub Search tool [52]. We gathered an initial total of 9,534 repositories by filtering according to the parameters presented in Table 3.1.

Table 3.1: Initial dataset for repository mining.

Number of Commits	2,000+
Number of Contributors	20+
Number of Stars	100+
Forks	Excluded
Requested On	2022-11-10
Number of Repositories	9,169

We set a minimum threshold for commits, contributors, and stars. This allows us to gather popular repositories with a good amount of contributions, indicating a reasonable size, history, and relevance. In addition, we clean the dataset by excluding fork projects both with the corresponding flag in GitHub Search and by manually removing those indicated as forks by the GitHub API. Finally, we remove projects that no longer exist (*e.g.,* those with an empty repository). In this preliminary study, we analyze a total of 9,169 projects.

After the cleaning process we get a lower number of projects due to inconsistencies between GitHub Search and the GitHub API. We consider the latter as the source of information about the status and nature of a project, using the former just as a pre-filter to start with a candidate set. An example of removed projects is *chinnkarahoi/jd_scripts*,[1] which appears in the GitHub Search dataset, but is no longer accessible (*i.e.,* disabled). Sometimes, GitHub resolves aliases of a requested project by default (*e.g.,* when a URL containing the aliased name is requested the main project is returned). In this case, we find that multiple aliases lead us to the same project, resulting in potential duplicates, and these aliases can also lead to forked projects. In our final dataset, we removed 23 projects that are no longer accessible and 8 fork projects. Finally, 3 projects failed to load due to memory issues, and we excluded them from the analyses.

For each project we retrieve the README files to extract the documentation sources that are present in their content. This approach allows to reach all the sources that the project owners deemed relevant and useful for different stakeholders willing to use or contribute to the project.

3.2.1 Sourcing and Retrieving the Data

The analyzed repositories, hosted on GitHub, are versioned using *git*,[2] a distributed version control system which we leverage to reconstruct the evolution of README files. *git* tracks the development history of a system via commits. Specifically, each commit keeps track of relevant information about which files were created, removed, and modified at a specific point in time (the commit's timestamp) by a developer (the commit's author). Via commits, it is possible to reconstruct different snapshots of the system.[3]

[1] See `https://github.com/chinnkarahoi/jd_scripts`

[2] `https://git-scm.com/`

[3] With respect to other versioning systems (*e.g.,* CVS, Subversion, Perforce), *git* keeps track of the changes between one version and the next by storing information about changed files as subsequent snapshots, as a mini filesystem [270].

There are five possible types of actions that can be performed on a file in a *git* commit:

- **Addition**: A new file is created and added to the *git* repository for tracking.
- **Deletion**: A file is deleted and ceases to exist in the *git* repository.
- **Modification**: An existing file is modified.
- **Rename**: An existing file is renamed without being moved to another path.
- **Move**: An existing file is moved to a different path. It has priority over a file rename in case the name of the file is also changed.

On top of commits, *git* also relies on the concept of branches and tags. A tag is used to mark a specific commit within the project. Typically, tags are employed to mark a release version of a system, often with some convention on the tag names (*e.g.,* semantic versioning[4]). A branch in a *git* repository represents an alternate timeline of the project. Usually a project has a stable default branch called `master` or `main` where all the features and changes are merged when ready. This default branch is the "main timeline" of the project, the one that presents only snapshots of the fully functioning system. At any point in time, another branch can be made, splitting from the main timeline and adding separate changes to the project. This branch can then be merged back into the stable branch when it is ready. Branches are often used to develop new features of the system by taking the current status (branching), developing and testing the feature (one or more commits), and updating the status of the main branch with the newly implemented feature (merging). Assuming this workflow, our analysis focuses on the main stable branch of each project and commits are the main medium that we use to extract the data of the evolution of specific files, and we can retrieve the version of a repository given a commit.

To extract this evolutionary data in a structured way, we must model the evolution of a README file and (re-)build its history.

3.2.2 Evolutionary Modeling

The evolution of the README files within a repository is represented by two entities (Figure 3.2):

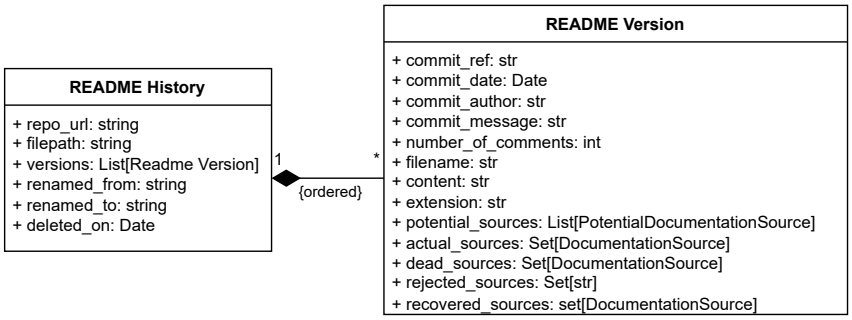


Figure 3.2: Evolutionary model for README files. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

[4]See `https://semver.org`

- **Readme History**: It contains an ordered list of Readme Version to represent the evolution of the README file and its path in time. It also contains data to know if the file was deleted on a certain date or whether it was renamed from or to another filename.
- **Readme Version**: It contains all the relevant data of the README file such as the date and commit hash of the version, the content of the file, and its documentation sources.

The notion of history is the main component of the model and, through each contained version, it allows to track the evolution of the file through time.

3.2.3 Building README Histories

Mining data for README histories is not as straightforward as it may appear. For each unique README file, we can get every version of such file and build its history. The path to the file is unique, but this same file may be renamed, moved to a separate directory, or deleted altogether and recreated with the same name later on, sometimes with a completely different structure and meaning, giving the "same" file multiple histories.

git provides information about file renames. For example, if a file is moved or renamed without changing its content and then committed, *git* will consider that as a rename operation. If the file is renamed or moved *while* its content is changed significantly in the same commit, *git* will consider the old file deleted and a new file added, also depending on the diff algorithm used.

To ensure that renames are properly tracked, the file move must be added to the *git index* (*e.g.,* `git mv [filename]`). The *index* is the staging area between the workspace and the repository, where all changes can be set up and combined before committing them. If a developer fails to add this move to the index prior to committing, the rename will not be detected.

We identify relevant commits as those which modify any README file in the repository. With these commits, we are able to build the README histories of the project. We keep track of commits deleting or renaming a README file to determine when a history ends and to support history chaining. The model in Figure 3.2 can thus be extended with the chained README History: The concatenation of the sub-histories, sorted by version dates. When we chain histories, we are checking whether two (or more) separate histories represent the same file. For instance, if `README.txt` is renamed to `README.md`, we have two histories for these two names that, in reality, represent the same file.

History Chaining

Multiple histories may pertain to the same file that was renamed and moved over time. There are four possible ways in which two histories can be temporally related to one another:

1. **Perfect Match**: When one history ends on the same commit where the other begins.[5]
2. **Separate**: When there is no overlap between the two histories.
3. **Contained**: When one of the two histories in a pair begins after and ends before the other.
4. **Overlapping**: When one history ends after the other has begun, excluding the cases in which one of the two histories is completely contained in the other.

We visually exemplify them in Figure 3.3.

[5]Perfect matches in date and commit hash are the typical case in which the file was renamed but *git* did not detect it. This is a specific case of overlap. We examine it separately as an optimization.

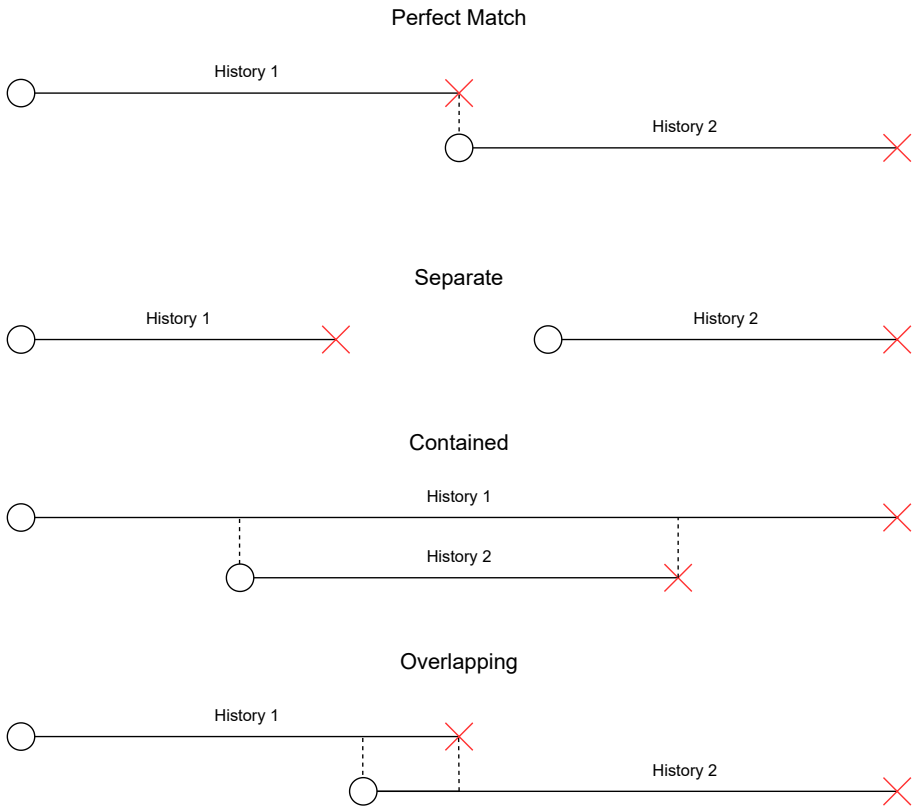


Figure 3.3: Possible chaining criteria. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

To address this issue we propose the chaining process described in Algorithm 1.

The chaining algorithm is computationally expensive. It iteratively chains pairs of histories, until no more chaining is possible. In particular, to be able to compare histories and find out whether they should be chained, we match every pair of histories that satisfies specific criteria.

The first step of the algorithm involves using the renames that *git* successfully identified. If a rename is confirmed to have happened between two histories, then we can be sure that they represent the same file, and we can chain these histories together.

Then the other criteria are checked in a specific order. When we look for pairs that are eligible for chaining, we must note that we check overlapping pairs only if no other criterion chained successfully. This is because, in the case of many histories that co-exist, one history is likely to overlap with many other unrelated histories. We often encounter projects with a large number of separate README histories that have co-existed at some point in time (*e.g.,* the Arduino project[6] by itself has 111 separate histories prior to chaining).

After obtaining the required pairs for each criterion, we compare versions of interest (*e.g.,* last version of the first history with the first version of the second history in a perfect match case) and we measure their similarity. Using the `diff` utility, as implemented in the `difflib` Python standard library,[7] if the similarity that we obtain is above a selected threshold, we merge two histories and create a unique chained history.

[6] `https://github.com/arduino/Arduino`
[7] `https://docs.python.org/3/library/difflib.html`

```
1: function Chain Histories
2:     INPUT: histories
3:     OUTPUT: chainedHistories
4:
5:     chainedHistories ← chainByRename(histories)          ▷ Use git renames for the first chaining
6:     hasPendingChaining ← True
7:     criteria ← [Perfect Match, Separate, Contained, Overlapping]
8:
9:     while hasPendingChaining do              ▷ Recursively chain until no further chaining is possible
10:        hasPendingChaining ← False                              ▷ Assume this is the last chaining
11:        for all criterion in criteria do              ▷ Calculate and chain pairs for each criterion
12:            pairs ← historyPairs(chainedHistories, criterion)
13:            intermediateChainedHistories ←
14:                  attemptChainingPairs(chainedHistories, pairs, criterion)
15:        end for
16:        if intermediateChainedHistories not equal chainedHistories then      ▷ Some chaining happened
17:            hasPendingChaining ← True                                  ▷ Trigger transitive chaining
18:            chainedHistories ← intermediateChainedHistories
19:        end if
20:    end while
21:    return chainedHistories
22: end function
```

Algorithm 1: Function Chain Histories. History chaining, merging different histories of the same file.

Once all potential chains have been checked, if at least one has been merged, we must now check for transitive chaining. Transitivity between a pair of chained histories manifests when each of the histories share a common sub-history. In this case, we can assume that both histories can be further chained together. Finally, the function `historyPairs` returns the pairs satisfying a specific criterion and the function `attemptChainingPairs` returns all the histories after chaining those matching the specific criterion passed as argument.

3.2.4 Documentation Sources

An important documentation source should be mentioned somewhere prominently for those interested to be able to find it. At the same time, README files are the first "landing point" when consulting the project's homepage on GitHub. So, we can assume that if a URL is important enough to exist in the README file of a project, then it could (and should) point to a relevant documentation source. But not all URLs are documentation sources (*e.g.,* link to a logo image).

In addition, we must consider the availability status of a potential documentation source, which can be either *alive* (reachable and still providing its content) or *dead* (*e.g.,* unreachable, moved, removed). Going back to our garden metaphor, a dead source is like a dead tree stump surrounded by ivy. It looks like a source, but it is just an empty shell, possibly the remains of a source which flourished in the past but that has no reachable content left.

We define an alive source as a URL that points to a source that responds to a HEAD or GET request and that could potentially be visited via a browser (*e.g.,* a URL that points to `https://example.com`). A dead source is instead a URL that does not respond to requests. The source is indeed dead when its URL is broken, but there is a possibility that it used to work at the time it was referenced. For this reason, we consider liveliness only for the latest snapshot of the documentation landscape, where a broken link corresponds to a problem in the landscape since the current README has unreachable sources. HTTP reachability is a condition that can be used to define the liveliness of most sources usually linked in README files.

Discord links, for example, provide an HTML page with meta-data on the community with the number of users, the number of online users etc. (see Section 4.7 for more details). In other cases, this limitation leads to the assumption that sources like an IRC channel are assumed to be alive if present in the README, without implementing further checks.

Parsing the README

To identify the documentation sources contained within a README file, we rely on Regular Expressions (RE) that identify patterns for URLs and non-URLs. The non-URLs in our case are e-mail addresses, IRC nodes, and filenames. REs discriminate between *potential* and *rejected* documentation sources (see Figure 3.4). The liveliness status of most potential sources is verified via HTTP checks to obtain actual *live* and *dead* sources.

We also maintain a record of the potential sources we automatically reject during the RE-based identification process. These *rejected* sources are then re-examined to check if any of them can be recovered. With this approach we try to recover sources resulting from truncated URLs and non-URLs whose reachability we cannot test via HTTP checks. For instance, a README file may reference a source file indicating only a path relative to the repository. Such source file is a documentation source that is rejected by the general REs. The recovery step allows us to include it as a *recovered source* with the proper destination.

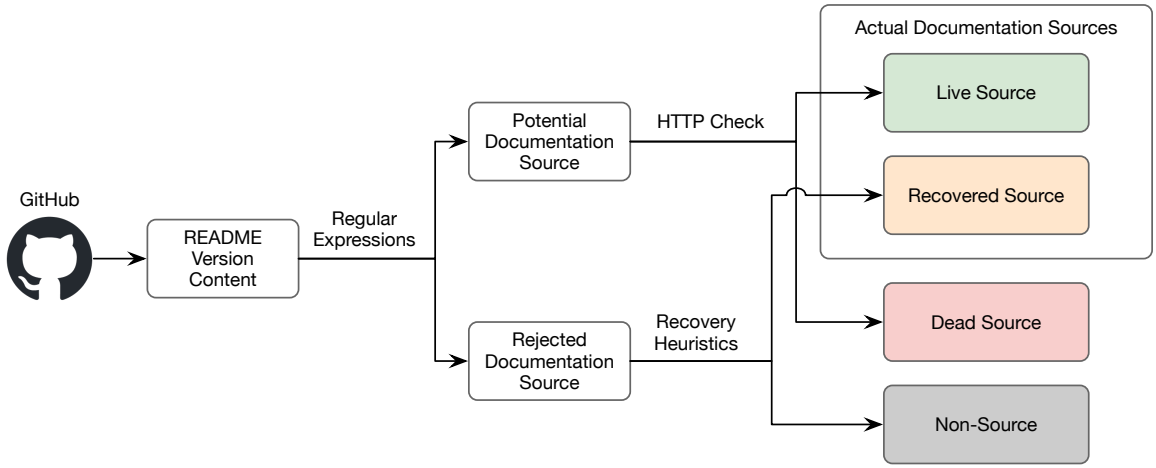


Figure 3.4: Process to detect documentation sources.

Manual Inspection

To refine the REs and capture the documentation landscape via the sources that compose it, we manually inspect a sample of the returned links, progressively refining the capturing expressions and adding specific expressions to the recovery step. We start with a sample set of 5 representative projects: `scikit-learn/scikit-learn`,[8] `arduino/Arduino`,[9] `apache/tomcat`,[10] `exoplatform/platform`,[11] and `pmd/pmd`.[12]

[8]See `https://github.com/scikit-learn/scikit-learn`
[9]See `https://github.com/arduino/Arduino`
[10]See `https://github.com/apache/tomcat`
[11]See `https://github.com/exoplatform/platform`
[12]See `https://github.com/pmd/pmd`

At each iteration we aim at fitting documentation sources in the current taxonomy, if we fail to find a suitable category, we extend and modify the taxonomy. In this step, we specifically focus on yet uncategorized cases. By focusing on what we failed to capture, we are able to extend and modify the taxonomy also according to the new found documentation source types. After each step, we disambiguate potential conflicts (*e.g.,* a category is too broad, too specific, overlaps with another category) in an open card-sorting way [232, 261]. When the conflicts are solved, the categories change accordingly. They can be split into multiple categories, merged into broader categories, and new categories can appear or disappear altogether.

The result of this manual analysis is a comprehensive taxonomy of the identified sources. In the following section we present the taxonomy and we briefly discuss each of its categories.

3.3 Taxonomy of Documentation Sources

We devised a taxonomy to describe the possible documentation sources constituting the documentation landscape of a software project. The root node is implicitly the abstract concept of Documentation Source. In Figure 3.5, we show the final resulting taxonomy.

For each source category (green) we have sub-categories (blue). Similarly, sub-categories can be further divided in a third layer (yellow). For each element of the taxonomy, we indicate in how many repositories we found a source of that category or sub-category. Each leaf sub-category in the hierarchy has its own heuristic to determine whether a source belongs to that category. When a category is detected, its own super-categories can be derived by following the hierarchy upwards. In some cases we focus on the top-level domain of the URL (*e.g.,* given `https://example.com/examples/myExample.html`, the top-level domain is `example.com`) to detect specific documentation sources, in others we consider the full URL (see Appendix B).

A common pattern in the taxonomy is that some sub-categories tend to have specific names of platforms (*e.g.,* WordPress) and a generic sub-category at the same level called *Custom [Category Name]* (*e.g.,* Custom Blog). Popular platforms are easily identifiable by their instance name and can be more easily described as a phenomenon, indicating a category. A less used, or project-specific platform (*e.g.,* a project's blog hosted on-premise on a private server) cannot be identified as a specific, shared medium, thus being referred to, in a more generic way, as a custom blog.

3.3.1 Conflicts Between Categories

The detection methods can identify the same documentation source as pertaining to multiple categories. To prevent inconsistencies and avoid conflicts, we assign priorities to rules based on their specificity. If methods for more than one category detect the source, the source is assigned to the category with higher priority (more specific).

We prioritize categories in the following order of decreasing priority (from most to least specific): Source File > Document > Relative File > Blog > Forum > Mailing List > Community Platform > Wiki > Repository-Related > Homepage > Uncategorized.

For example, if a documentation source is detected by Source File, Document, and Repository Related, we prioritize the assignment of the Source File category as it is the highest in the priority list. If no category is successfully recognized, the source is Uncategorized.

We developed heuristics for categories at different levels (*e.g.,* top-level, sub-category). Specificity still determines the assigned category. The most specific category detected in the hierarchy is assigned as the source type.

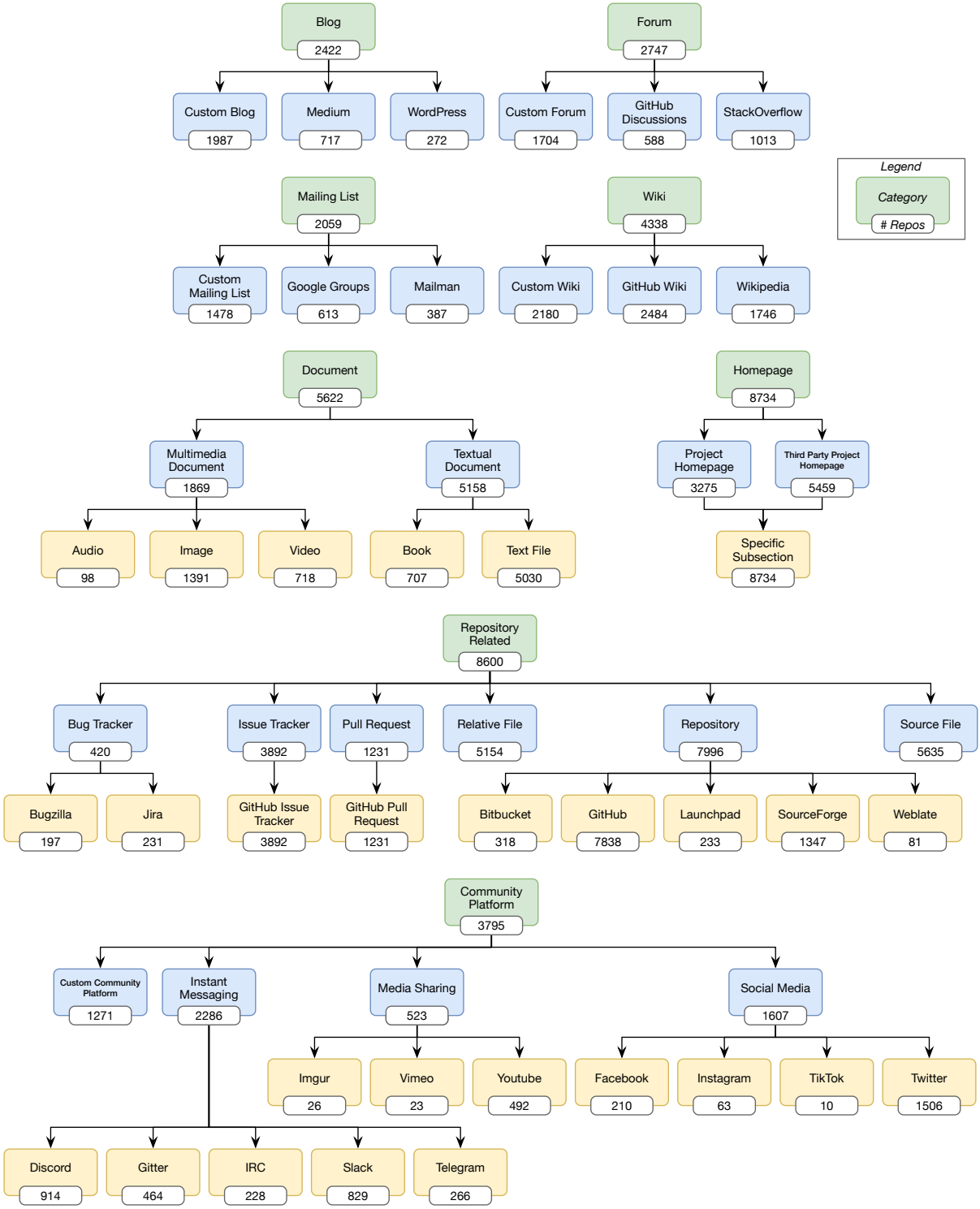


Figure 3.5: Taxonomy of the software documentation landscape.

3.3.2 Categories of the Taxonomy

Blog

A Blog, short for weblog, is an informational website. It typically contains a series of posts in reverse chronological order [271], with a diary-like structured content. The Blog category has three sub-categories: Custom Blog, Medium, and WordPress (Figure 3.6).

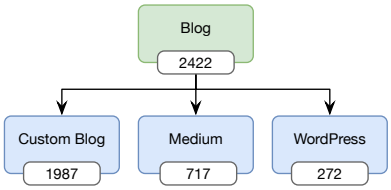


Figure 3.6: Taxonomy of the Blog category.

Custom Blog: A blog that is either self-hosted or that exists under its own specific domain that does not fall under any of the previous categories.

Medium: A blog that is hosted under the Medium domain.[13] Medium is an online publishing platform launched in August 2012, and is regarded as a blog host [272].

WordPress: A blog that is hosted under the WordPress domain.[14] WordPress was originally created as a tool to publish blogs [273] but evolved in a web content management system.

Community Platform

A Community Platform encompasses platform types that share the way that documentation is generated and consumed on them. This processes tend to be strongly crowd-sourced and very volatile. The key point is that users are the ones providing the documentation. We found four sub-categories of the Community Platform category: Custom Community Platform, Instant Messaging, Media Sharing, and Social Media (Figure 3.7).

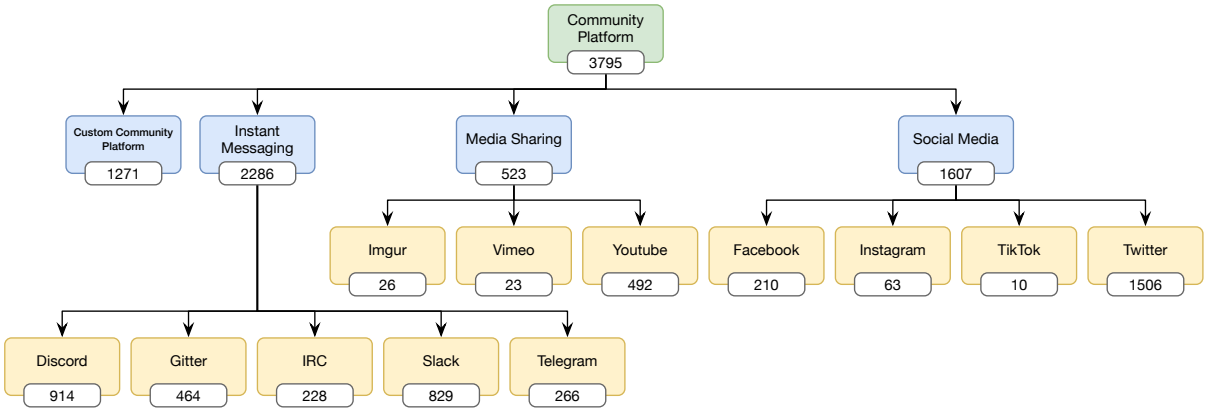


Figure 3.7: Taxonomy of the Community Platform category.

Custom Community Platform: A community platform that does not fall under any of the other sub-categories. For instance, a project may have its own custom community portal with its own login page that does not adhere to any other widespread platform.

[13] See `https://medium.com/`
[14] See `https://wordpress.com/`

Instant Messaging: A form of online communication that offers real-time text transmission over the internet. The user of an instant messaging platform (often through a specific application) plays a crucial role in generating documentation that is highly volatile.

Instant messaging provides a way to quickly ask and answer questions without imposing any a-priori structure. Conversations can change topic in a matter of seconds and, given the tendency to give more saliency to recent messages, newer discussions can bury old ones. The Instant Messaging category has five sub-categories: Discord, Gitter, IRC, Slack, and Telegram.

Discord: An instant messaging and Voice over Internet Protocol (VoIP) platform [274].[15]

Gitter: An open source instant messaging and chat room system[16] for developers and users of GitLab and GitHub repositories [275].

IRC: A reference to an Internet Relay Chat server or channel. IRC is a text-based chat system for instant messaging [276].

Slack: A cloud-based freemium[17] cross-platform instant messaging service.[18]

Telegram: Telegram Messenger[19] is a globally accessible freemium, cloud-based and centralized instant messaging service [277].

Media Sharing: Platforms where a user mainly shares media files such as videos or images. We identify three sub-categories corresponding to three popular instances of media sharing platforms: Imgur, Vimeo, and YouTube.

Imgur: An online image sharing and image hosting service [278].[20]

Vimeo: A video hosting, sharing, and services platform [279].[21]

YouTube: A video sharing and social media platform [280].[22]

Social Media: Social media websites and applications enable users to create content of different types (*e.g.,* written posts, images, videos), and to share them with a list of contacts that forms a social network. The key feature is the ability to share content with a large audience, also given the algorithms that these platforms often use to suggest potentially relevant content to users not in the social network of a content creator (thus allowing to reach a new audience in a pool as big as the platform's popularity). This is in contrast with other sources that are more focused either on the discussion aspect (*e.g.,* Forum) or on specific types of content (*e.g.,* images on Instagram, videos on YouTube).

Under the Social Media category, we found four sub-categories, again corresponding to four popular social platforms: Facebook, Instagram, TikTok, and Twitter.

Facebook: A social media and social networking service [281].[23]

Instagram: A photo and video sharing social networking service [282].[24]

TikTok: A short-form video hosting service [283].[25]

Twitter: Currently rebranded to X, a social media and social networking service [284].[26]

[15] See `https://discord.com/`
[16] See `https://gitter.im/`
[17] Free but with features that can be unlocked by paying for the premium tiers.
[18] See `https://slack.com/`
[19] See `https://telegram.org/`
[20] See `https://imgur.com/`
[21] See `https://vimeo.com/`
[22] See `https://www.youtube.com/`
[23] See `https://www.facebook.com/`
[24] See `https://www.instagram.com/`
[25] See `https://www.tiktok.com/en/`
[26] See `https://x.com`

Document

The Document category encapsulate *files* that are meant to be human-readable or human-understandable. Sources in this category range from simple text files to complex formats such as specific markup files, PDFs, or multimedia documents. Document has two main sub-categories: Textual Document and Multimedia Document (Figure 3.8). Textual Documents can be Books or Text Files. Multimedia Documents can be Audios, Videos, or Images.

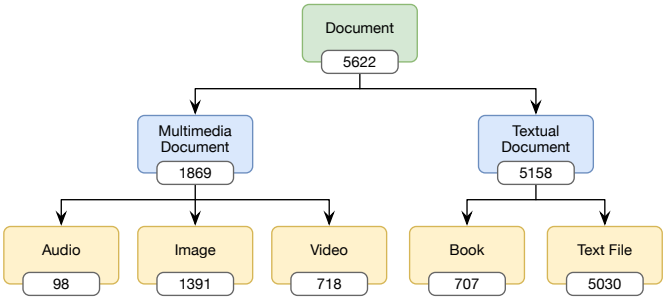


Figure 3.8: Taxonomy of the Document category.

Multimedia Document: Document containing more than one of the children types media such as audio files, images, and or videos at the same time.

Audio: An audio file.

Image: An image file.

Video: A video file or a link to a specific video on a platform.

Textual Document: Document including text files from simple ASCII .txt files to proprietary formats (*e.g.,* .docx for Microsoft Word text files). The text file must be human-readable (potentially with the aid of specific conversion devices (*e.g.,* .epub for e-readers). We found two sub-categories of Textual Document: Book and Text File.

Book: A source that references a physical book. While a physical book cannot be directly accessed via a URL, a website where a book can be purchased gives access to the book.

Text File: Any other Textual Document source that references a text file.

Forum

A Forum is a web-hosted site with message organization functionalities. Similar to community platforms, most of the content is typically crowd-sourced, with users being able to create a thread of discussion around a given topic. It distinguishes itself as its own category due to the combination of crowd-sourced content and persistency of the generated documentation.

The threaded structure of a forum, along with the typical threads persistence across years (as long as the website is hosted) are the key aspects of this category. We found three sub-categories for Forum: Custom Forum, GitHub Discussions, StackOverflow (Figure 3.9).

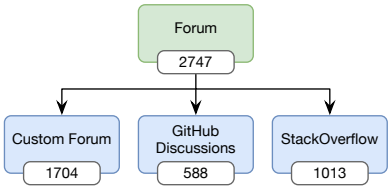


Figure 3.9: Taxonomy of the Forum category.

Custom Forum: A forum that has its own structure, often loosely based on templates but not directly recognizable as a specific forum application (*e.g.,* a custom phpBulletinBoard [285]).

GitHub Discussions: A forum for collaborative communication among the community of a GitHub project [286]. We group here forums directly hosted on GitHub and accessible through the Discussions tab of the project.

StackOverflow: online question and answers (Q&A) website[27] characterized by the Q&A structure for its threads. StackOverflow can still be considered a forum. The ability to vote on the questions and answers provides crowd-validation to both and allows fine-grained curation of the source contents.

Homepage

Homepage is a documentation source that points to the home page of a project-related website. We distinguish between two types of Homepages: Project Homepage and Third-Party Project Homepage (Figure 3.10). Both represent the same type of source in terms of content and structure. The difference between one or the other is their relevance to the project they reference and the one they are referenced by. As a practical example, the `scikit-learn` project referencing their own personal website `https://scikit-learn.org/` would fall into the Project Homepage category. If the same website is referenced by another project that makes use of `scikit-learn`, the source would fall under the Third-Party Project Homepage category.

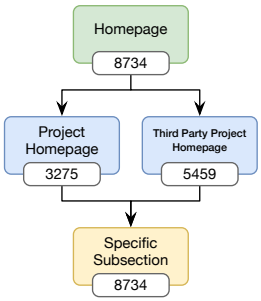


Figure 3.10: Taxonomy of the Homepage category.

This category of the documentation landscape is characterized by the lack of crowd-sourced content. A homepage is typically maintained and curated by its owners, often at least partially overlapping with the codebase developers, while a user can only consult the source without being able to participate in its generation.

Both sub-categories of Homepage have a shared common child: The Specific Subsection category. We specify this since the homepage of a website is not always directly referenced and, instead, the URL of the project website points to a location at a deeper path. These are specific sub-sections under the homepage domain and serve the purpose of giving direct access to a source more specific for the context in which it is mentioned.

Mailing List

A mailing list is a computer-based communication system that distributes e-mails to a predefined group of recipients. It operates as a centralized platform, where copies of messages submitted by authorized users are delivered to each subscriber in the list.

[27] See `https://stackoverflow.com/questions`

Collaborative interaction happens through e-mails, leading to persistent crowd-sourced documentation in e-mail archives. Mailing List has three sub-categories: Custom Mailing List, Google Groups, and Mailman (Figure 3.11).

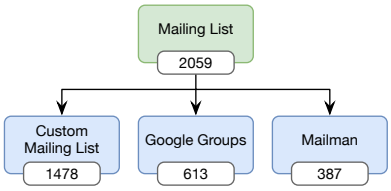


Figure 3.11: Taxonomy of the Mailing List category.

Custom Mailing List: A source that references a mailing list that is not supported by any popular mailing list application (*e.g.,* `mailman`).

Google Groups: Google service that provides discussion groups for people sharing common interests.[28] Accessible online with a forum-like interface or through a mail client as a traditional mailing list. Following Google's acquisition of Deja's Usenet archive in 2001, Google groups started to merge and blur the boundaries between forums, mailing lists, and Usenet discussion groups [287].

Mailman: A free software for managing e-mail and e-newsletter lists.[29]

Repository-Related

Repository-Related is a category for documentation sources that describes aspects internal to the project inner workings (*e.g.,* the development lifecycle). It has six sub-categories: Bug Tracker, Issue Tracker, Pull Request, Relative File, Repository, Source File (Figure 3.12).

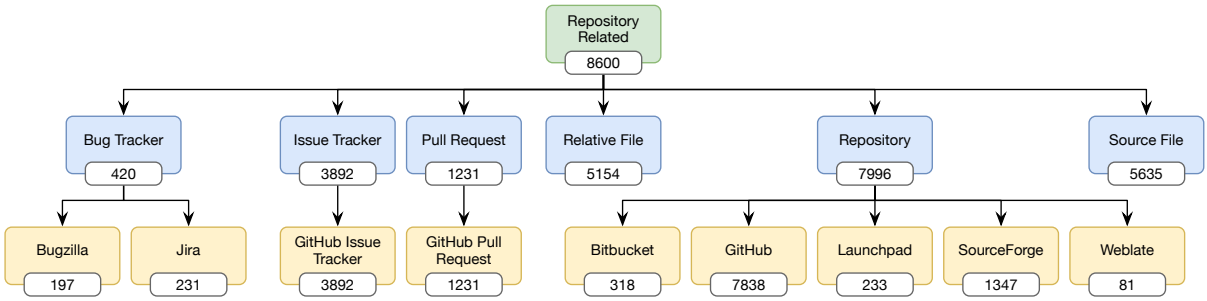


Figure 3.12: Taxonomy of the Repository Related category.

Bug Tracker: A tool that documents bugs by allowing developers to report them during the development or maintenance of a project. This category may be regarded as a less generic type of Issue Tracker, as it pre-dates issue tracking systems, which also focuses on new features requests and general discussions about issues with the tool, not strictly bugs. We identify two sub-categories for Bug Tracker: Bugzilla and Jira.

Bugzilla: A web-based general-purpose bug tracking system and testing tool [288].[30]

Jira: An issue tracker developed by Atlassian [289].[31]

[28] See `https://groups.google.com`
[29] See `https://list.org/`
[30] See `https://www.bugzilla.org/`
[31] See `https://www.atlassian.com/software/jira`

Issue Tracker: Tool that documents "issues", to notify project maintainers and developers of problems, bugs, and new requested features, keeping track of their evolution. Under Issue Tracker, we specify the GitHub Issue Tracker as its only sub-category.

Pull Request: Thread-like conversation about changes pushed to a branch in a repository on GitHub. Contains discussions, code reviews, and follow-up commits before merging into the base branch [290]. Under Pull Request, we include only the sub-category GitHub Pull Request.

Relative File: A type of file that is referenced via a relative path existing in the project repository, and that we cannot identify to be a Source File or a Textual Document.

Repository: A documentation source that holds the source code and other (versioned) artifacts of a software project. We identify five sub-categories of Repository: Bitbucket, GitHub, Launchpad, SourceForge, and Weblate. These instances are project repository aggregators, home of multiple projects, referenced as documentation for dependencies and related projects.

Bitbucket: A *git*-based source code repository hosting service owned by Atlassian [291].[32]

GitHub: A platform and cloud-based service for *git*-based collaborative software development and version control, owned by Microsoft [292].[33]

Launchpad: An online platform that allows users to develop and maintain software [293].[34]

SourceForge: A web service[35] oriented to management of OSS projects [294].

Weblate: An open source web-based localization tool[36] with version control [295].

Source File: A file that contains source code (different from a generic Textual Document and Text File). Sometimes directly referenced when higher level documentation is not available, or when specific implementation details are discussed in other forms of documentation.

Wiki

A Wiki is a website that allows collaborative modification of its content and structure directly from the web browser. The key feature of wikis that distinguishes it from other sources is the ability for any user to edit the content (sometimes with authorization credentials, *e.g.,* to avoid content trashing [296]). This makes a wiki the product of a collaborative effort from the community rather than a more authoritative source curated by few. Under the Wiki category, we identify three sub-categories: Custom Wiki, GitHub Wiki, and Wikipedia (Figure 3.13).

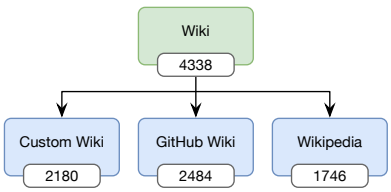


Figure 3.13: Taxonomy of the Wiki category.

Custom Wiki: A wiki that is not hosted on any widespread wiki platform.

GitHub Wiki: A wiki on GitHub.

Wikipedia: A free-content online encyclopedia[37] written and maintained by volunteers [297].

[32] See `https://bitbucket.org`
[33] See `https://github.com/`
[34] See `https://launchpad.net/`
[35] See `https://sourceforge.net/`
[36] See `https://weblate.org/en-gb/`
[37] See `https://en.wikipedia.org/`

3.4 Visualization of the Documentation Landscape

We presented all the categories of our taxonomy of the documentation landscape. The preliminary analysis consists of progressive refinement of documentation sources in a dataset of 9,169 GitHub projects until all the potential sources in the README files have been captured by a set of REs, based on specificity-driven priorities. We aimed to keep the categories specific enough to be able to capture platforms that distinguish themselves in their characteristics, while keeping them as broad as possible to abstract irrelevant differences. We tried to strike a balance between specific instances of platforms (*e.g.,* Discord) and general categories (*e.g.,* Blog), preferring platform instances whenever the popularity of a platform resulted in a specific pattern of inclusion in the documentation landscape of a project.

With the defined taxonomy, we can visualize snapshots of the documentation landscape and capture its evolution, both for specific open source projects and for the landscape as a whole. For more details regarding the implementation of the miner we used for this study, RagnaDok, please refer to Appendix A and to the Master Thesis of Tommaso Rodolfo Masera [196].

To explore the mined data, we devised and implemented two different visualizations for README histories and the documentation landscape. The former is a polymetric view[38] of the sources present in README histories and their contents over time. The latter offers project-specific and aggregate views of the software documentation landscape and its sources.

3.4.1 README History Visualization

Given a GitHub project, we can visualize its histories, versions, and documentation sources over the whole history of the project's repository. In Figure 3.14, we show an example of the visualization of a landscape's evolution and highlight the different parts constituting the user interface (UI) of the frontend.

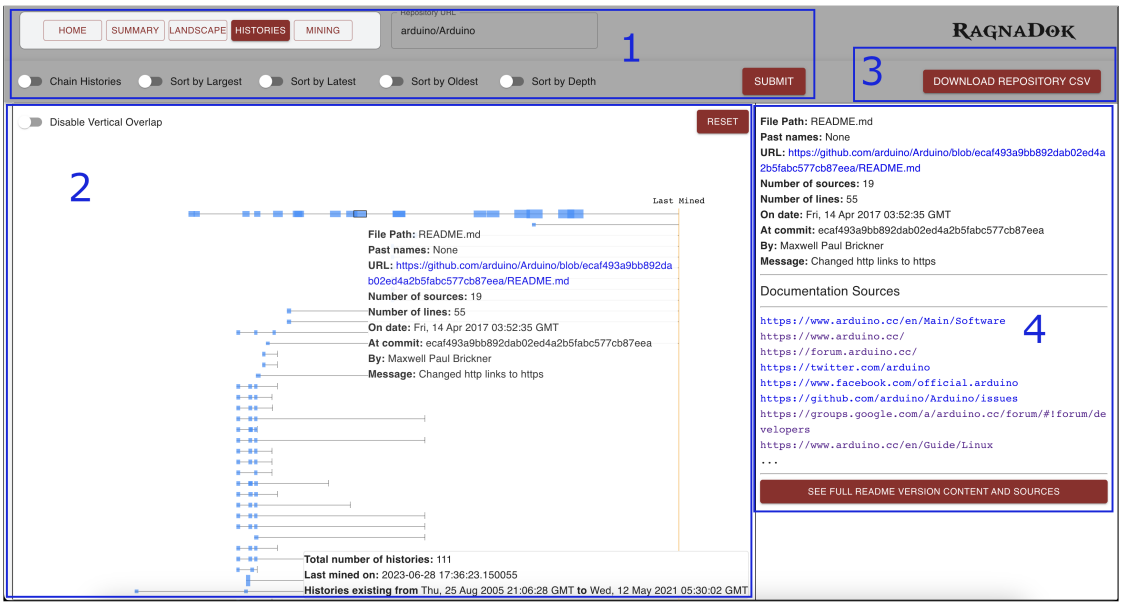


Figure 3.14: Visualization of README histories. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

[38] A software visualization technique that enriches glyphs representing the entities under study with software metrics information mapped onto properties of the glyphs (*e.g.,* size, shape, color) [131, 132].

The main UI components are:

1. **Option bar**: It allows to select previously analyzed repositories and to customize the visualization (*e.g.,* sorting the histories, visualizing chained or unchained histories).
2. **View**: The view is a Scalable Vector Graphics (SVG) image of the histories. Each line corresponds to a history and each rectangle corresponds to an individual README version. The width and height of the rectangle indicate its number of documentation sources and its number of lines, respectively. Rectangle sizes are normalized and have a minimum size to show a README version even if empty. On hover, rectangles show insights about the version they represent.
3. **Extraction for processing**: It is possible to download the documentation sources in the latest version of the project for further analyses.
4. **README version information**: Displays information about the selected version and the first few documentation sources identified in the README. It allows navigating to the complete list of sources and the README contents via the button at the bottom.

Then we have the version information visualization. It shows the contents of the currently selected version and its sources (see Figure 3.15 for an example). Its main UI components are:

1. **Utility buttons**: They allow to view the rendered README file directly on GitHub and download an annotation CSV for the specific version.
2. **Documentation source panel**: Displays the identified documentation sources of this README version, separated by liveliness status.
3. **Version contents panel**: Shows the raw contents of the README file version.

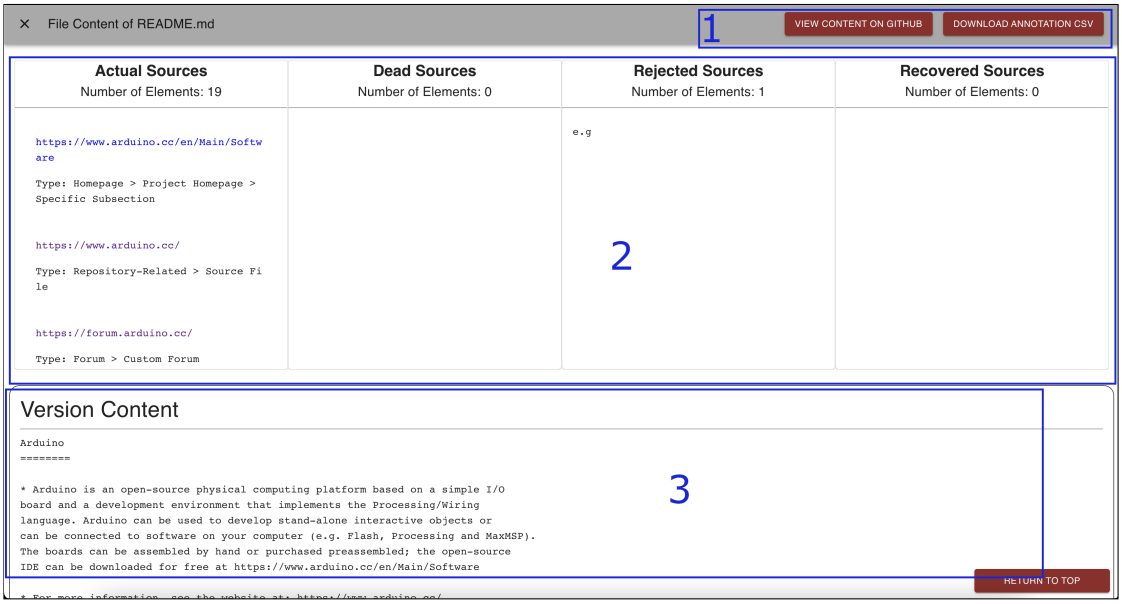


Figure 3.15: Visualization of a README Version. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

3.4.2 Documentation Landscape Visualization

Visualizing information about the README and the sources is fundamental for inspection, but to really grasp the shape of the landscape, and even more so to understand its evolution, we leverage a visualization based on a customized treemap with a fixed layout. Treemaps are a space-filling method of visualizing large hierarchical data sets [117] that optimizes space usage and can scale for visualizing hierarchical and recursive large structures in limited 2D spaces [24, 25, 73, 99, 215, 216, 228, 256]. We use the taxonomy we defined in Section 3.3 for fixing the layout with all the possible sources. Each category of the taxonomy represents a potential documentation source type that we map onto a rectangle in the visualization.

We visualize the documentation landscape of a single project and the aggregate landscape across all the mined and analyzed projects. They share the same layout with a few differences in the way their data is grouped and displayed (*e.g., data binning*[39], color scale).

Documentation Landscape of a Project

For a single project, we visualize the landscape by mapping the number of documentation sources per category on color. We use buckets with the following ranges to distinguish among qualitatively different situations of the sources presence: 0 (no sources), 1 (exactly one source), 2 to 5 (a few sources), 6 to 10 (some sources), more than 10 (many sources). We choose these ranges as they outline whether a source is present or absent in a project. Figure 3.16 show an example of this visualization for the Elasticsearch project.

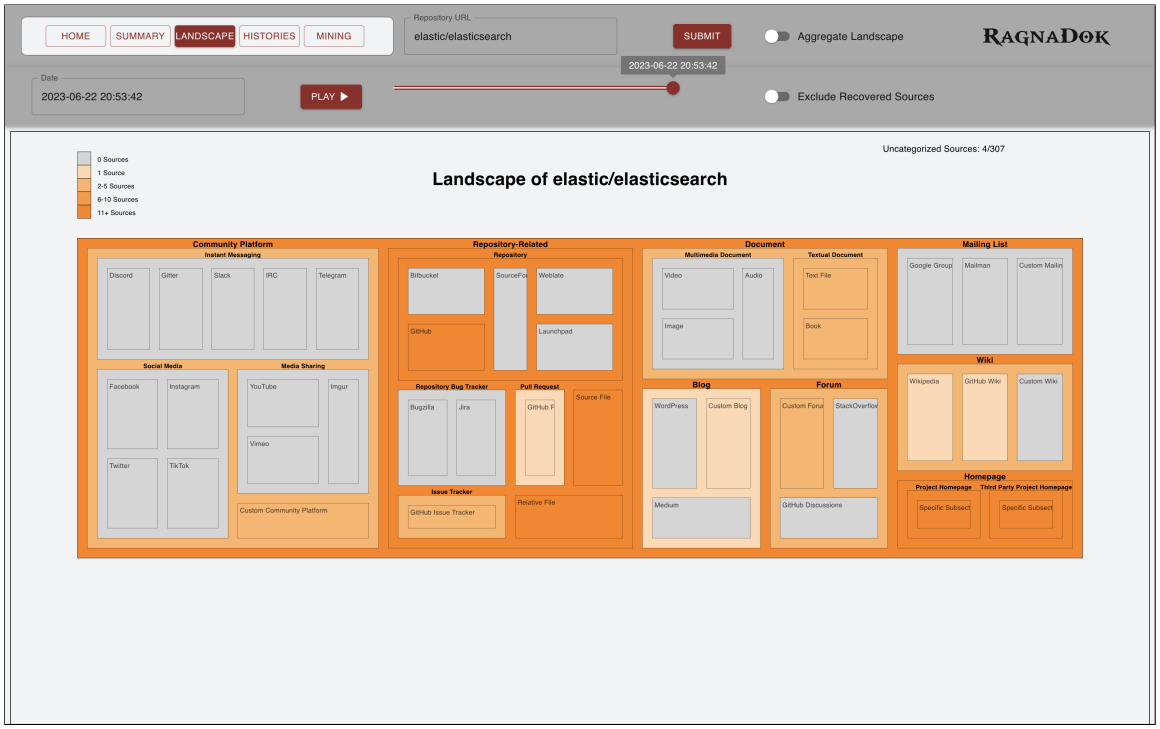


Figure 3.16: Documentation landscape of the Elasticsearch project (`elastic/elasticsearch`) in RagnaDok. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

[39] A data pre-processing technique used to reduce the effects of minor observation errors [298], in our case it contributes to produce visual stability for minor changes, thus highlighting phenomena of relevance.

The landscape can be viewed at any commit date (selectable by the corresponding date picker and slider) that modified a README file in the repository. The Play button shows the animated evolution of the project's landscape by visualizing each version in sequence, with a short delay between two consecutive versions.

The custom treemap features a legend of the used colors corresponding to the selected bins. We also show the number of documentation sources that we did not categorize within our taxonomy. A treemap is a space-filling method of visualizing large hierarchical data sets [117]. In the example visualization (Figure 3.16), we can see how the latest version of Elasticsearch[40] that we have mined contains many documentation sources in the GitHub, Source File, Relative File and Homepage categories. Other categories instead show with a lower number of occurrences.

Aggregate Documentation Landscape

We visualize the aggregate documentation landscape by focusing on the overall presence of a documentation source among all the analyzed projects (Figure 3.17). We bucket the data differently, compared to the previous view, skewing the bucketing towards being more sensitive for smaller percentages (*i.e.,* more buckets in the lower half of the range). We have one bucket every 10th percentile until 50%, after which the documentation source is so popular that we utilize larger thresholds of 75% and 90%. We use a white-to-black color scale like in a heat-map, where white indicates absence of a source (exactly 0%, no repository has sources of that category)[41] and black indicates that more than 90% of the repositories contain at least a source of that category.

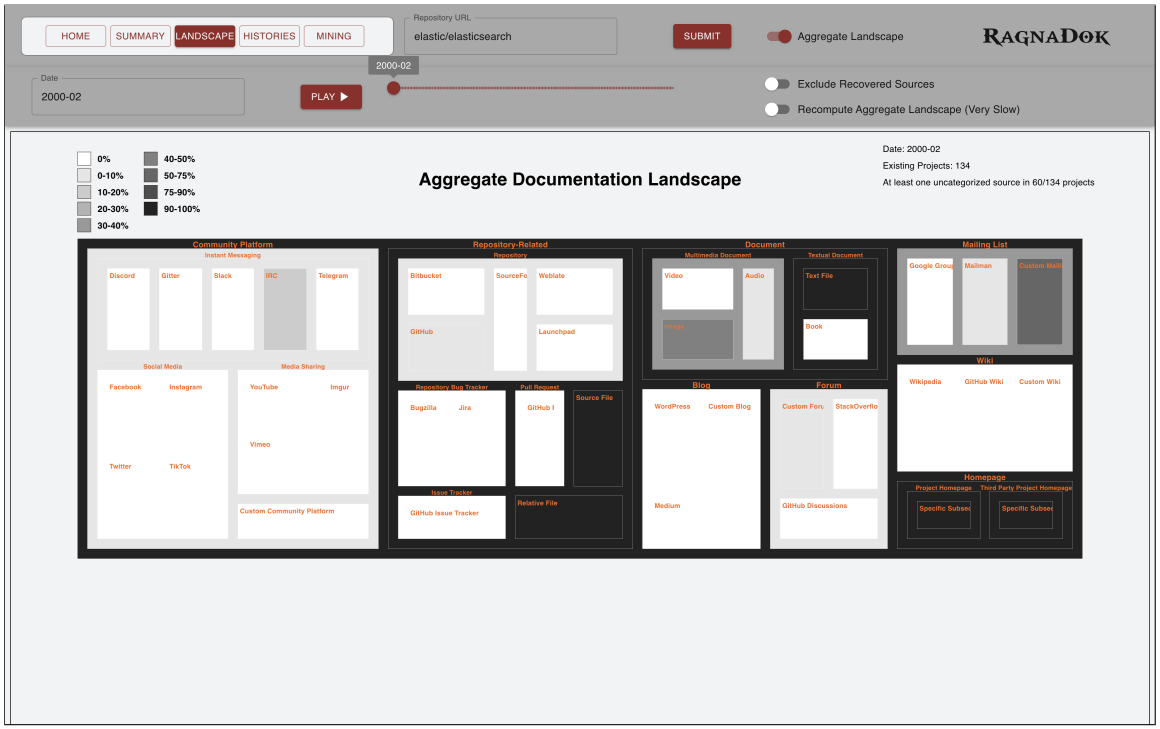


Figure 3.17: Aggregate documentation landscape in RagnaDok. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

[40] `https://github.com/elastic/elasticsearch`

[41] The interval 0%–10% is the only one open on both sides, with 0% as a separate bucket.

The view maintains the same layout of the single project documentation landscape view. It shows the number of existing projects at the presented date and the number of projects in which our taxonomy fails to capture at least one documentation source. The aggregate landscape tends to vary less over short periods of time. Because of this and the computational cost to map the whole landscape over the complete dataset at any point in time, we compute and display the aggregate landscape every 3 months.

3.4.3 Dataset

We summarize statistical information about the mined projects and their histories (see Table 3.2).

Table 3.2: Statistical information about the 9,169 mined repositories and their histories.

Metric	Total Sum	Average	Median	Minimum	Maximum
# of Histories	808,291	88.1	16	1	52,889
# of Sources	4,000,553	436.1	121	0	103,626
# of Source Categories	178,017	19.4	18	0	53

We can see how, despite the high average of 88.1 histories per repository, the median is 16, still indicating a complex evolution of one or more files of interest. There are some outliers that present an extremely high number of histories (Figure 3.18).

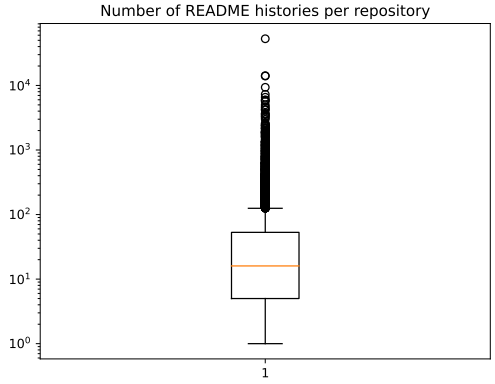


Figure 3.18: Box plot of the number of README histories per repository. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

Outlier Projects

There are outliers with a number of histories that range between the thousands and the tens of thousands. Among these, we inspect three repositories (Table 3.3). Two of them do not contain a single software system that evolved over time, instead, they are information aggregators.

The *demisto/content*[42] repository contains an ever-growing collection of playbooks, automation scripts, report templates and other useful information provided by Demisto[43] to automate and orchestrate security operations.

[42]See `https://github.com/demisto/content`

[43]Now Cortex XSOAR Platform, a SecOps platform to automate security operations, targeted at large scale security operation centers.

Table 3.3: Outlier projects.

Project Name	# Histories	Contents Brief Description
`demisto/content`	13,986	List of playbooks, automation scripts, report templates and other useful content to automate and orchestrate security operations [299].
`doocs/leetcode`	52,889	Multi-language solutions for problems from LeetCode, "Coding Interviews (2nd Edition)", "Coding Interviews (Special Edition)", "Cracking the Coding Interview (6th Edition)", etc. [300].
`qmk/qmk_firmware`	14,168	Firmware for various types of keyboards [301].

The *doocs/leetcode*[44] project is a repository that holds solutions in many programming languages (*e.g.,* Java, Python, C++, Go, TypeScript, Rust) to programming problems from the LeetCode[45] programming learning platform. Each of the solutions contains its own README file, thus inflating the number of histories.

These two project use GitHub as storage for much more than a single software system, thus justifying being identified as outliers with respect to the number of histories. The third project is *qmk/qmk_firmware*,[46] a slightly different single codebase software project: The firmware for multiple keyboards. The large number of histories is given by the `keyboards` directory in the repository. It holds information about all the supported keyboard models, each with its own dedicated README file, with a result similar to the previous two examples in terms of histories.

These outliers are inevitable in a fully automated analysis like the one presented in this preliminary study. Nevertheless, in the future, more effort should be devoted to curating the dataset of the aggregated landscape. Outlier detection on the number of histories and sources could help identify potentially problematic repositories and those not related to a single system.

3.5 Preliminary Analysis Discussion

We provide an analysis of the data we mined and processed with RagnaDok [196]. We present a number of case studies to look in-depth at the software documentation landscapes of specific projects and give examples of what the landscape looks like. In Figures 3.19, 3.20, and 3.21, we show the comprehensive evolution of the aggregate documentation landscape over the last two decades, as obtained from all sources in the projects in our dataset. In Figure 3.22, we show a zoomed-in view on the same snapshots of the Community Platform category.[47]

The evolution of the aggregate documentation landscape from the year 2000[48] shows how the eight top-level categories evolved over time (see Figure 3.23).

We can see that homepages and repository-related documentation sources closely follow both the trend and the magnitude of the increasing number of considered projects.

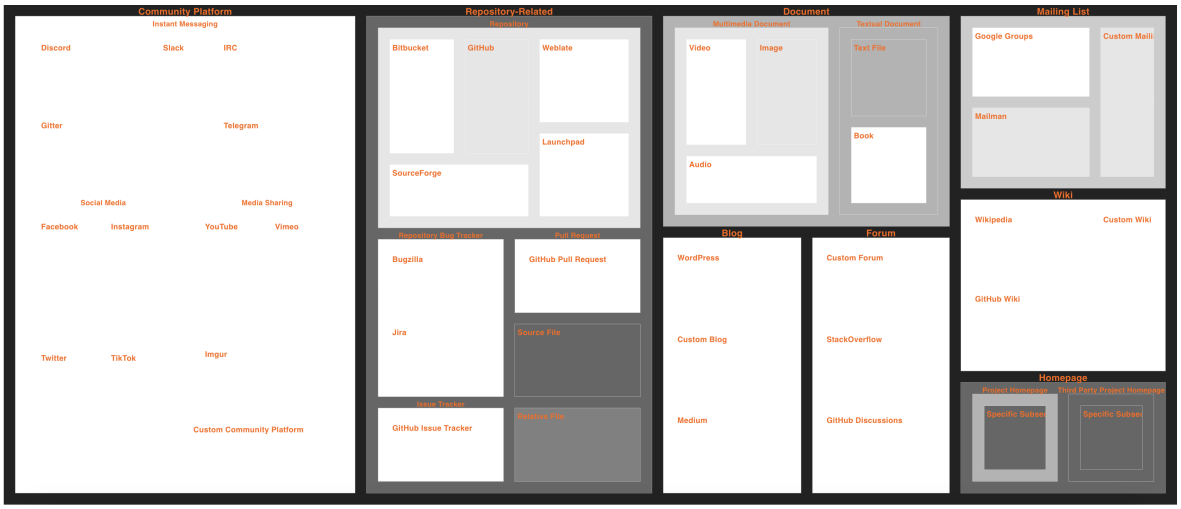
[44] See `https://github.com/doocs/leetcode`

[45] See `https://leetcode.com/`

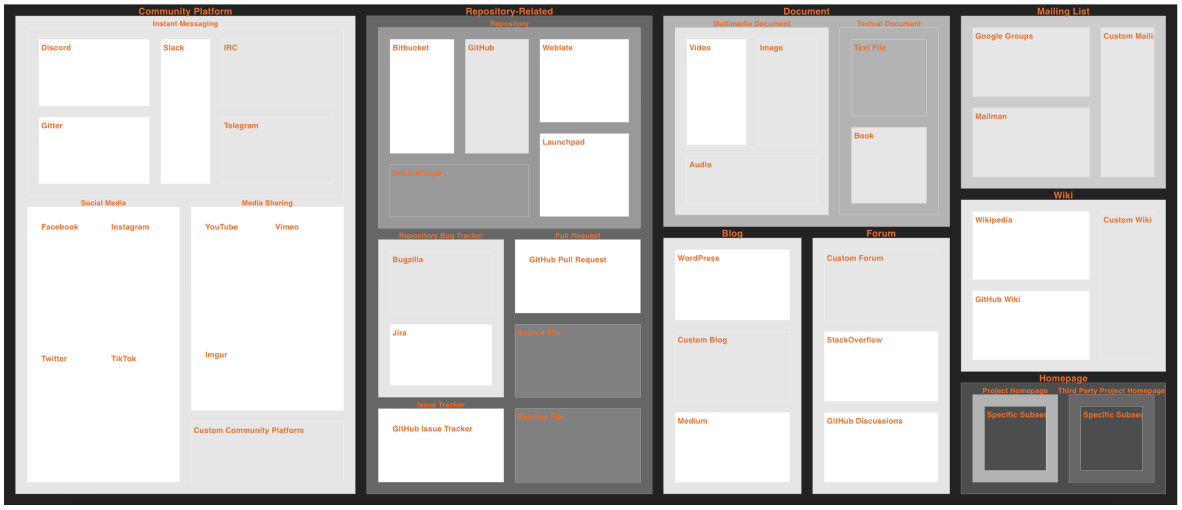
[46] See `https://github.com/qmk/qmk_firmware`

[47] See the following breakdown in different periods for details about misclassifications of sources (*e.g.,* the appearance of a Telegram source in 2003) and its relationship with the detection methods described in Appendix B.

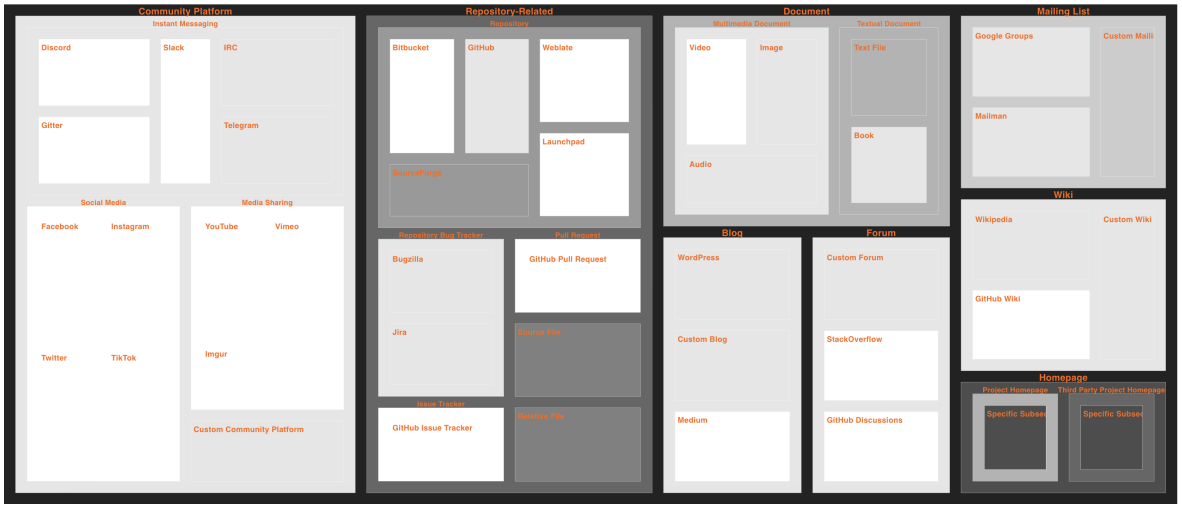
[48] The number of existing projects before the year 2000 is relatively low, less than 150, to draw any meaningful insight. Moreover, such projects, have been added to GitHub after having their histories converted from previous versioning systems.



(a) Aggregate documentation landscape in 2000.

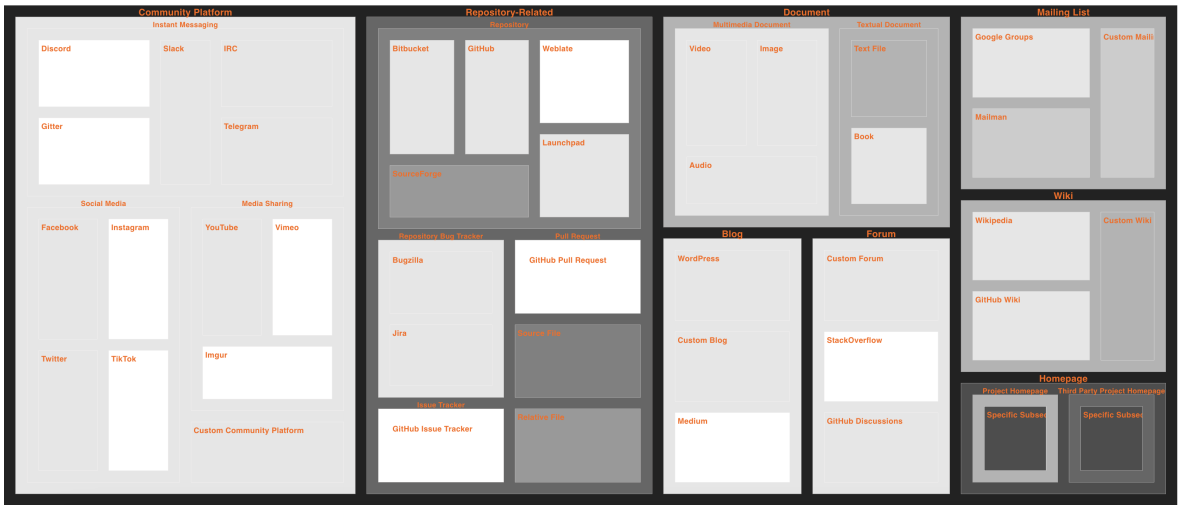


(b) Aggregate documentation landscape in 2003.

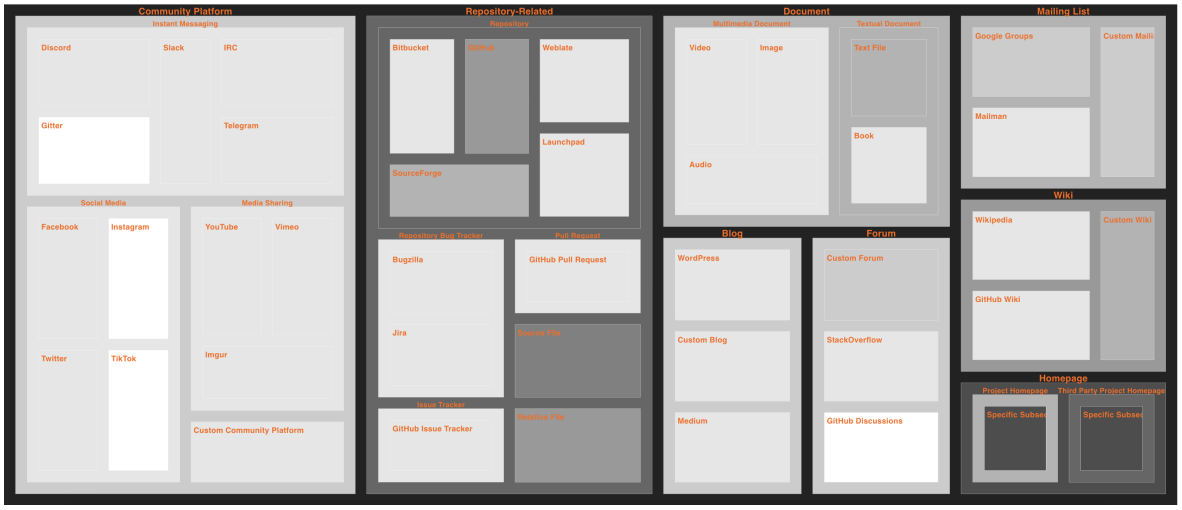


(c) Aggregate documentation landscape in 2005.

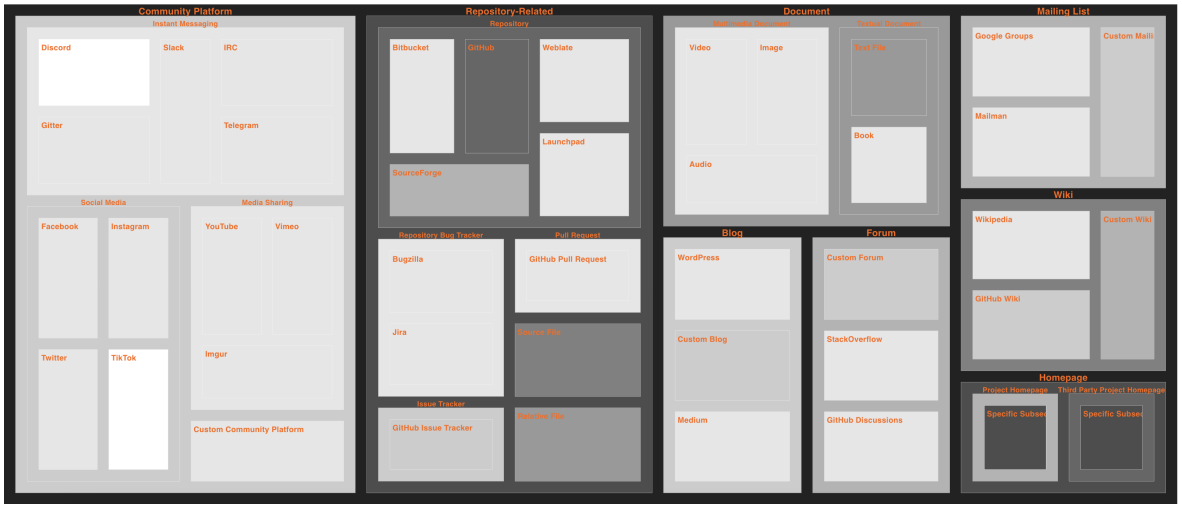
Figure 3.19: Aggregate documentation landscape between 2000 and 2005. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].



(a) Aggregate documentation landscape in 2009.

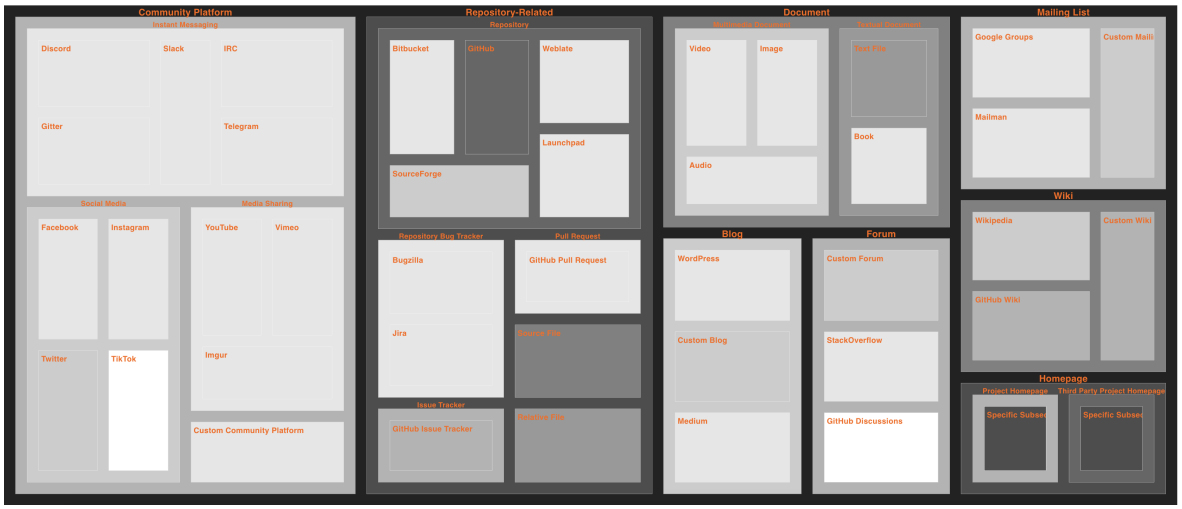


(b) Aggregate documentation landscape in 2012.

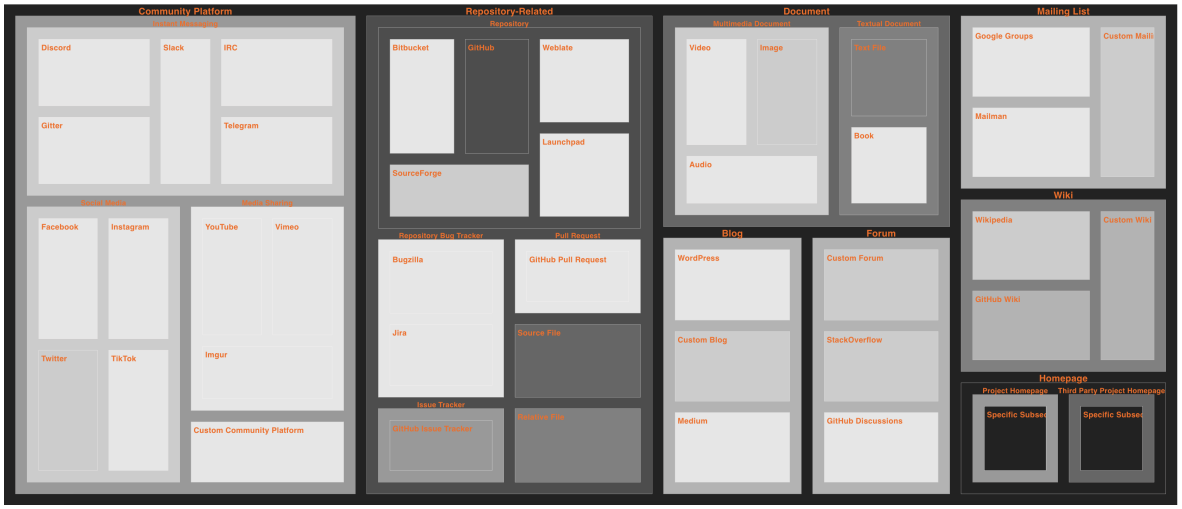


(c) Aggregate documentation landscape in 2015.

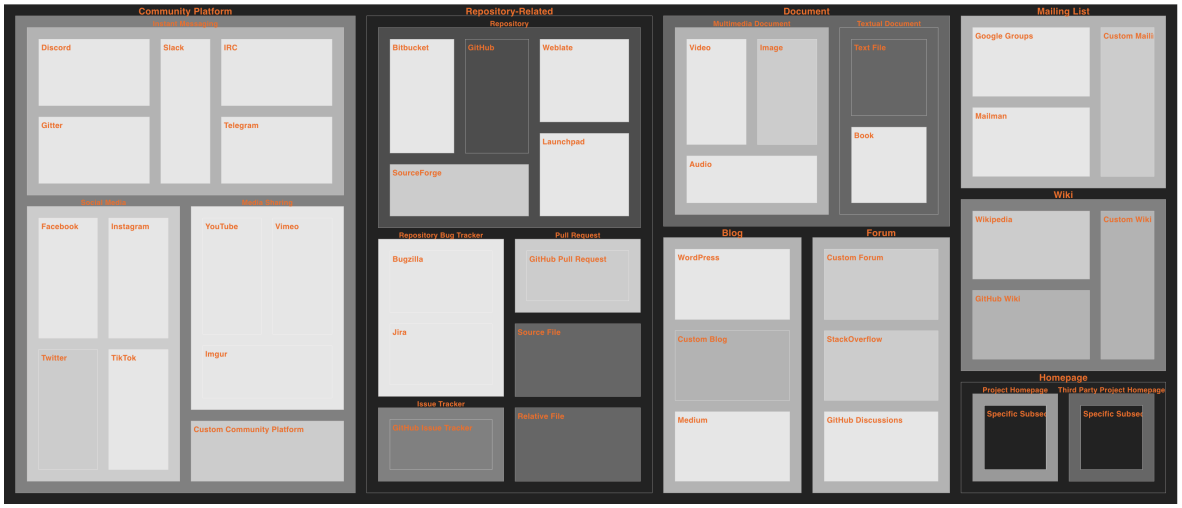
Figure 3.20: Aggregate documentation landscape between 2009 and 2015. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].



(a) Aggregate documentation landscape in 2016.



(b) Aggregate documentation landscape in 2019.



(c) Aggregate documentation landscape in 2023.

Figure 3.21: Aggregate documentation landscape between 2016 and 2023. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

Figure 3.22: Aggregate evolution of the Community Platform category between 2000 and 2023.

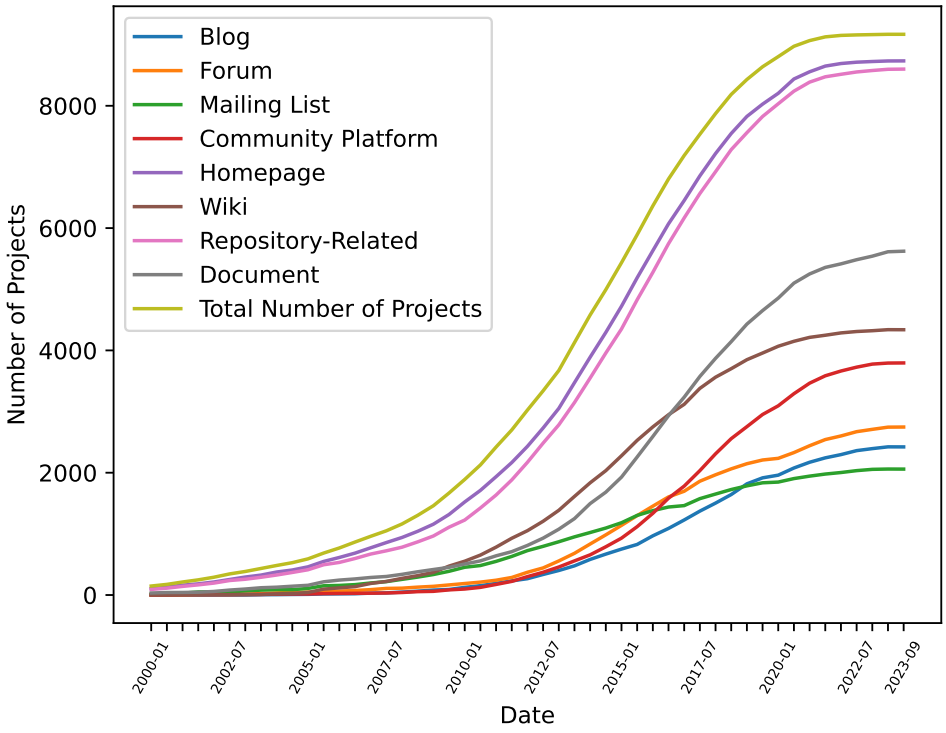


Figure 3.23: Presence in projects of the eight top-level categories of the taxonomy (and total number of projects). Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

Most of the projects reference a homepage, whether it is external or internal, as well as some documentation source that is tightly related to the repository itself (*e.g.,* cross document references). We also see how the categories with newer forms of documentation sources (*e.g.,* Community Platform) rise sharply in recent years (between 2013 and 2020[49]), while other initially popular categories are progressively overcome by other platforms. This is the case of the Mailing List category, surpassed by forums, community platforms, and blogs in the last decade.

> *The documentation landscape constantly changes. By effect of new additions and new categories, new sources substitute the old ones, rising and falling in popularity. But the landscape also grows in size and complexity, with more and more sources to form the modern snapshot of the landscape.*

Early Years

The documentation landscape begins growing during the first five years of the 2000s. With an increasing number of projects being created, the documentation landscape begins to expand under a twofold effect. The percentages of Table 3.4 show, by normalizing them with respect to the total number of considered projects in the same year, how categories not even present in 2000 (*e.g.,* Forum and Wiki) reach a relative presence of 8% and 7% respectively. More stable, although still growing, references to documents, homepages, and the repository itself, denoting an evolution of good practices instead of a revolution.

[49]Where we see an effect of the filtering criteria to select the initial dataset. The number of projects remains almost stationary after 2020, determining a plateau in most of the already widespread documentation sources.

Table 3.4: Status of the aggregate landscape in its first five years (2000, 2003, and 2005).

Year	**2000**	**2003**	**2005**
Number of projects	**147**	**385**	**591**
Category	**Percentages**		
Blog	0.0%	0.3%	1.9%
Community Platform	0.0%	2.3%	2.9%
Document	22.5%	25.5%	26.9%
Forum	0.0%	3.4%	8.1%
Homepage	72.1%	76.4%	78.0%
Mailing List	10.2%	17.1%	18.3%
Repository-Related	62.6%	67.0%	69.9%
Wiki	0.0%	1.3%	6.9%
Uncategorized	27.9%	22.3%	22.0%

It is worth noting that community platforms and blogs also start to appear in 2003 and 2005, although with a very low frequency (2.3% and 1.9% respectively). In 2000, the main platforms of communication are mailing lists. They almost double in relevance in the following 4 years.

To complement the evolution of the categories presented in Table 3.4, we can see the detailed evolution between 2000 and 2015 (see Figure 3.19 and Figure 3.20).

In 2003, SourceForge (in the Repository sub-category) joins the Homepage, Document, and File categories as a widespread documentation source, appearing in most of the projects. Mailing lists become more popular as Google Groups and Mailman are used. The first community platforms appear along with the first blogs, forums, and wikis. Bugzilla starts being referenced for bug tracking and reporting. Surprisingly, we also seem to find Telegram in the documentation landscape, but, upon manual inspection, this is due to a misclassification of some links in the early landscape. In 2005, we find the appearance of the first WordPress and Wikipedia documentation sources, together with Jira, while most of the other sources remain stable in popularity.

Towards Modern Sources

The documentation landscape continues to grow steadily between 2005 and 2015, populating most of the layout defined by our taxonomy, with a few exceptions (*e.g.,* Gitter, Instagram, and GitHub Discussions until 2012; Discord and TikTok until 2015). As the number of projects increases, so does the number of sources (see Table 3.5). Some categories, like for example Mailing List between 2012 and 2015, start decreasing in their relative usage. This may be due to a combination of new projects not adding them to their landscape anymore and older projects substituting them with newer documentation categories (*e.g.,* community platforms).

In this time period, the evolution of the top-level categories gives limited information, besides confirming the continuation of some previous trends (*e.g.,* blogs and community platforms becoming even more popular). With the aid of the detailed and progressive visualization of Figure 3.20, we can instead highlight some interesting phenomena. In 2009, we begin seeing social media and media-sharing websites on the rise, with YouTube, Twitter, and Facebook showing up as documentation sources referenced in the READMEs. The Repository sub-category becomes more popular, with Launchpad, Bitbucket, and GitHub wikis starting to appear. We also find another false positive for Slack, as it did not exist yet at this time.

Table 3.5: Status of the aggregate landscape between 2009 and 2015.

Year	**2009**	**2012**	**2015**
Number of projects	**1,460**	**3,339**	**5,434**
Category	**Percentages**		
Blog	5.6%	10.1%	13.8%
Community Platform	4.1%	11.1%	17.1%
Document	28.6%	27.9%	35.5%
Forum	9.7%	13.3%	21.0%
Homepage	79.3%	81.8%	86.9%
Mailing List	23.0%	23.8%	21.8%
Repository-Related	66.1%	74.4%	80.1%
Wiki	25.3%	36.1%	41.9%
Uncategorized	21.6%	22.3%	29.7%

Moving forward to 2012, we see how community platforms and media-sharing websites start occupying more of the landscape. Google Groups follow along, and so does GitHub. Vimeo and Imgur make an appearance together with Medium and StackOverflow.

GitHub now appears in a large percentage of projects at this point in time, and some of them have a GitHub wiki and GitHub pull requests. References to the same project and cross references between projects become common practice. Again, our RE present another inconsistency, since Discord did not exist in 2012, very likely indicating another false positive.[50]

Finally, in 2015, we see how social media become more relevant, with the addition of Instagram to the landscape. While instant messaging remains stable, we see how now Slack, Gitter, and Telegram do exist in the landscape, this time as legitimate additions. As GitHub takes over other repository-related and collaborative development platform categories, we notice how SourceForge slightly decreases in usage. Blogs are also on a constantly increasing trend since their appearance in 2003. Bugzilla and Jira are steadily replaced by the GitHub Issues tracking system with its bug tracking and feature request capabilities (see Figure 3.20c).

The Current Landscape

We analyze the last stretch of this evolution, until the current snapshot of the aggregate documentation landscape. Between 2015 and 2023 some older documentation sources have declined and have been overtaken. The landscape almost completely covers our taxonomy now (Figure 3.5).

We can see a resemblance of stability in the evolution trends of the landscape between 2018 and 2023 (Table 3.6). This is just an apparent stability, resulting from obtaining the taxonomy from the current snapshot of the repositories and the large bins chosen for the different colors in the visualization. Most of the sources are well established and the top-level categories do not change much, still with some shifts in relative importance (*e.g.,* Community Platforms and Instant Messaging), but we can see how many documentation sources are into the process of being (or have been) replaced (see Section 4.7 for a more detailed and fine-grained study on this evolution). For instance, we see Bugzilla and Jira almost disappear from the landscape supplanted by issue tracking from GitHub Issues, which has replaced older bug trackers, with GitHub dominating the Repository category (see Figure 3.21).

[50]Note that the apparent incidence of false positives is magnified by the choice of buckets and the difference between 0% and 0%–10%, highlighting a change in categories even if a single false positive is found.

Table 3.6: Status of the aggregate landscape between 2018 and 2023.

Year	**2018**	**2021**	**2023**
Number of projects	**6,363**	**8,430**	**9,169**
Category	**Percentages**		
Blog	15.2%	21.6%	26.4%
Community Platform	21.0%	32.7%	41.4%
Document	40.7%	52.6%	61.3%
Forum	22.8%	25.5%	30.0%
Homepage	88.5%	92.8%	95.3%
Mailing List	21.7%	21.2%	22.5%
Repository-Related	82.9%	89.7%	93.8%
Wiki	43.2%	45.7%	47.3%
Uncategorized	34.1%	45.7%	51.8%

Forums and wikis are the only categories where GitHub co-exists with other proprietary (*e.g.,* StackOverflow for the Forum category) and custom (*i.e.,* for the Wiki category) alternatives.

A note about a potential threat to the validity of this conclusion is represented by the specific choice of projects we analyzed. By focusing on GitHub projects, the chances of GitHub being overrepresented in the dataset for its imminence as the project's hosting platform should be taken into account.

In Figure 3.24, we show a stacked bar chart of the percentages of repositories in which we found at least one source of the specified top-level category.

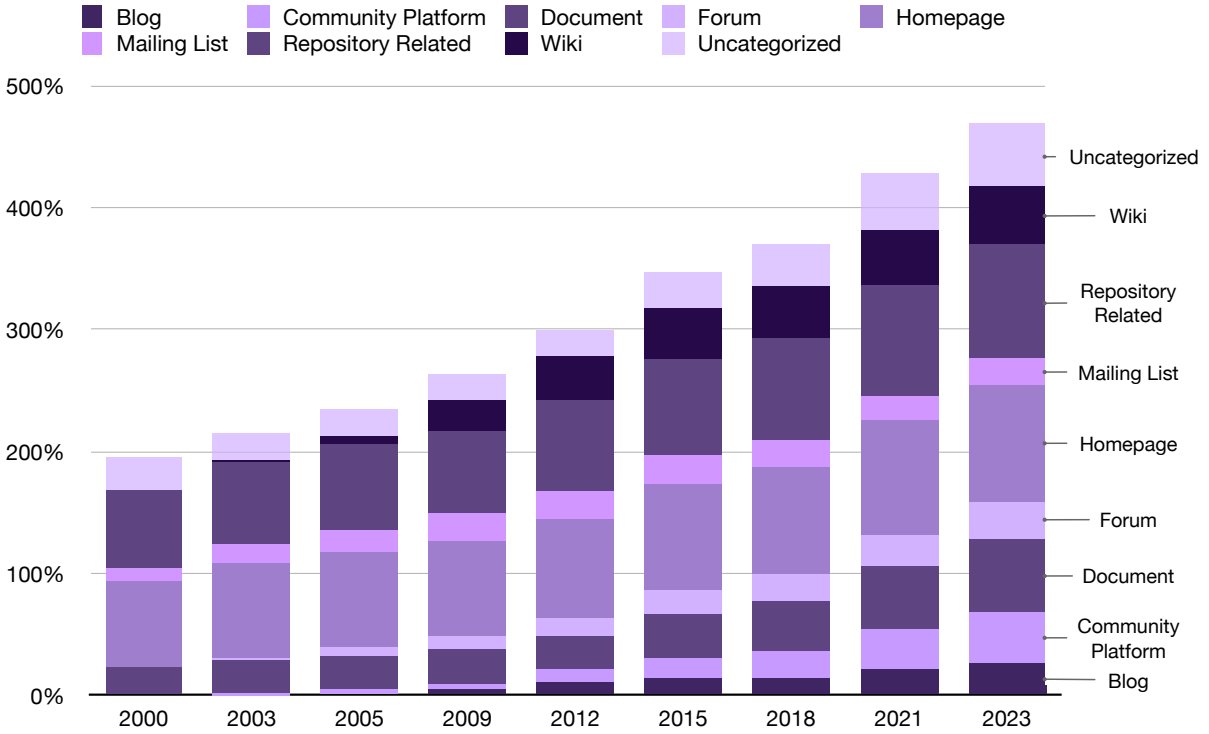
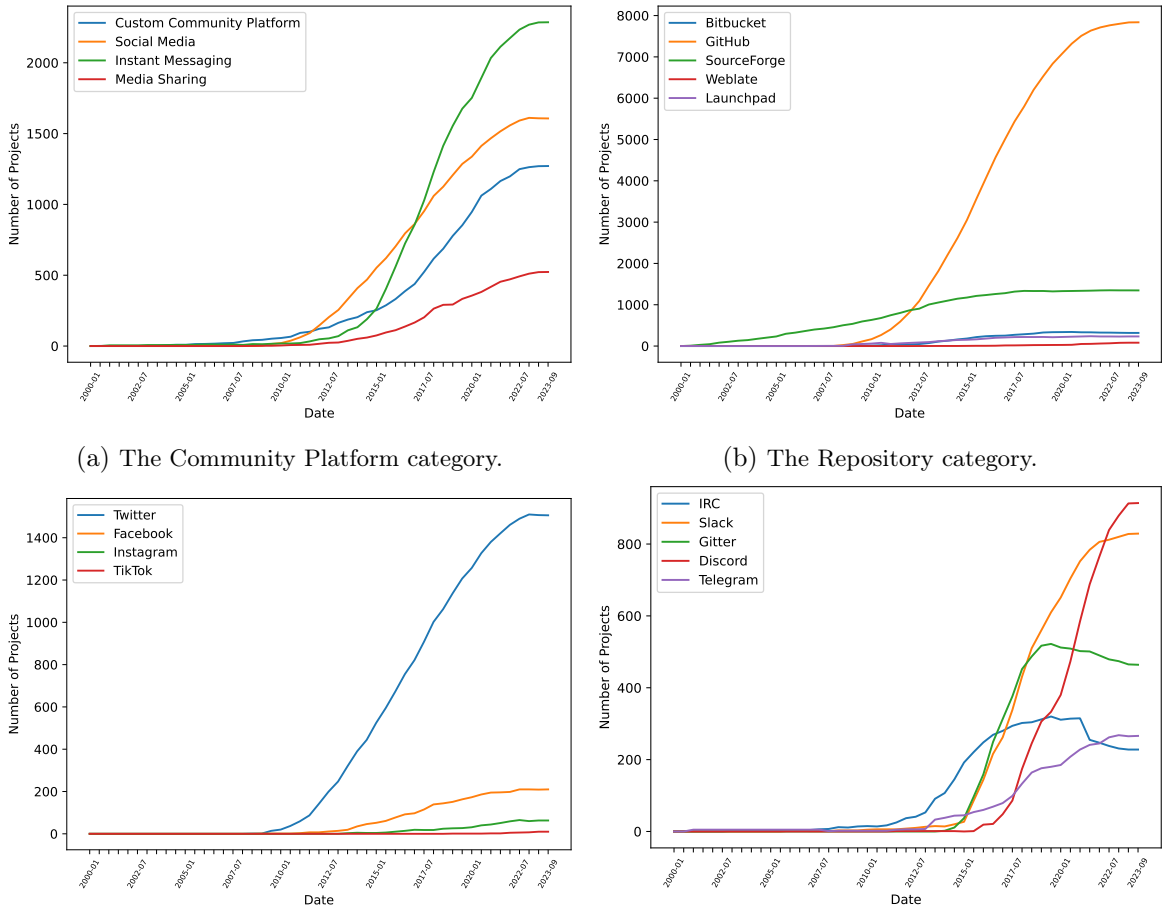


Figure 3.24: Evolution of the top-level categories. Aggregate documentation landscape 2000–2023.

Complementing the previous analyses, we can see how the landscape grows over the last two decades, in terms of number of covered top-level categories (from 4 in 2000, to full coverage since 2012). The landscape also becomes more homogeneous, while some categories, for example mailing lists, after a period of growth (2009–2012) remain stable or even decrease their presence in recent years. Community platforms, while not as widespread as homepages and repository-related categories, become the fourth most widespread category. To understand this evolution more in detail, we analyze some of the categories and sub-categories.

3.5.1 Documentation Landscape Evolution: Categories vs. Sub-Categories

We analyze the evolution of specific categories and sub-categories of the software documentation landscape (Figure 3.25). We start by comparing the evolution of the four sub-categories in the Community Platform category (Figure 3.25a).



(a) The Community Platform category.

(b) The Repository category.

(c) The Social Media category.

(d) The Instant Messaging category.

Figure 3.25: Documentation landscape of sub-categories of the Community Platform, Repository, Instant Messaging, and Social Media categories. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

The Instant Messaging category explodes in popularity after 2015. The increase in the use of Custom Community Platforms is similar, although less steep. The presence of multimedia sharing platforms, while never mainstream, have become non-negligible in the last 5 years. Overall, instant messaging is the community platform favored by developers, followed by social media.

Media sharing platforms are trailing behind. Software development has shown a growing interest towards developing a community around collaborative projects and these platforms are the tool of choice to foster this collaboration.

Similarly, we analyze some leaves of the taxonomy, under different sub-categories, and how they evolve in peculiar and interesting ways. For the Repository sub-category, we can see how GitHub towers over other repositories (Figure 3.25b). Since its appearance, it has become more and more popular, leaving the others behind.[51]

In Social Media (Figure 3.25c), Twitter is the favored platform, also considering that the number of captured Twitter documentation sources is under-representative due to the lack of a RE to identify Twitter handles (*e.g.,* `@username`—see Twitter's *detection method* in Appendix B). Surprisingly, Facebook has been found mentioned (either with groups or specific pages—more similar to relevant user profiles or for project news dissemination) in more than 200 projects.

Finally, we see how also instant messaging platforms show a very interesting scenario. The popularity of older platforms, like IRC, is in decline. Similarly, but more surprisingly, the relatively recent Gitter, with its GitHub integration[52] has seen a decrease in its usage.

On the contrary, newer, feature-rich, and more popular platforms, like Discord and Slack, have been on the rise in recent years, quickly overtaking their competitors (Figure 3.25d). Apparent attrition for the adoption of Telegram, with respect to the similar IM platforms, might be worth investigating more in detail, since the presence of the platform in more than 200 projects in our already restrictive dataset, is far from being irrelevant.

3.6 Case Studies

We conclude this chapter with 3 case studies, showing various interesting shapes that the documentation landscape of a project can take. Going back to the garden metaphor of Section 3.1, some of the following projects present an empty garden when they are created. Whether they grow into intricate, overgrown, yet florid jungles to navigate with a machete or minimalistic, sleek, and functional, zen gardens, or still remain barren wastelands around an empty house, these examples prove the usefulness of analyzing a single project landscape. They shed light on how, besides the value of analyzing the evolution of the software documentation landscape as a collective phenomenon, each project can benefit from reasoning on the status of its landscape (and potentially compare it to a healthy baseline).

> *This work just scratched the surface of the possibilities of defining landscape types and characterizing them in terms of the health status of the documentation of the corresponding projects. In the future, we envision deifining documentation landscape stereotypes to compare against in order to gain insights on the status of a project's landscape and to develop improvement strategies.*

3.6.1 A Jungle Garden: `scikit-learn`

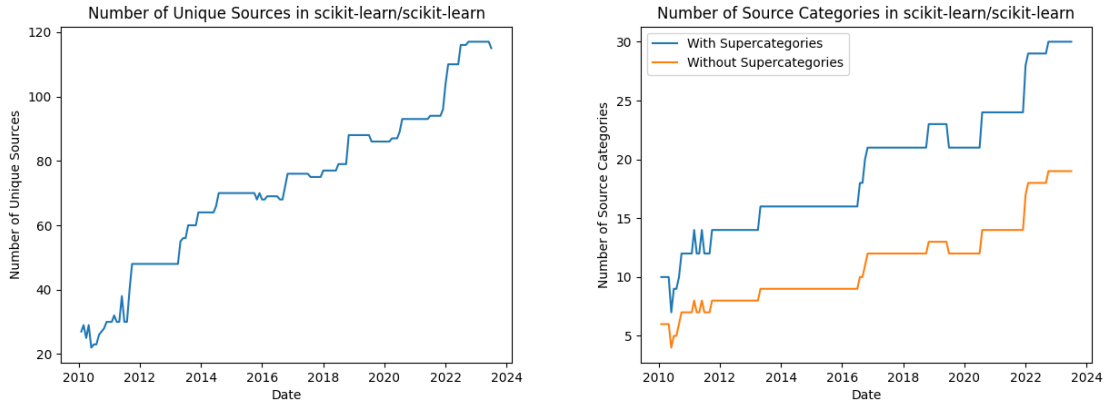
The Scikit-Learn project[53] is a popular open source Python framework that provides a comprehensive collection of tools for machine learning data science-oriented tasks. With this case study, we aim to show how the documentation landscape of the project evolves and presents a rapid (sometimes chaotic) growth in documentation sources in recent years.

[51] See the previous threat for a potential dataset driven bias about this dominance.
[52] The use of GitHub credentials and the linking of issues in the chat via `#<issue_number>` handles.
[53] See `https://github.com/scikit-learn/scikit-learn`

Overview

At the time of mining, the project has 70 README histories and 30,155 commits. Its latest snapshot features 74 unique documentation sources in 28 different categories. We capture most of the landscape via identification of live and dead sources.

The scikit-learn library has more than a decade of GitHub history. It presents a gradual but constant growth in the referenced documentation sources (Figure 3.26a). Furthermore, the landscape already presents documentation sources at the beginning of the project, with 7 categories and more than 20 sources in one of the very first versions in 2010. The project shows a rather steep growth in the leaf sub-categories in the last 3 years (Figure 3.26b).



(a) `scikit-learn`: Unique documentation sources. (b) `scikit-learn`: Unique source categories.

Figure 3.26: Documentation landscape over time of `scikit-learn`. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

The Initial Flowerbeds

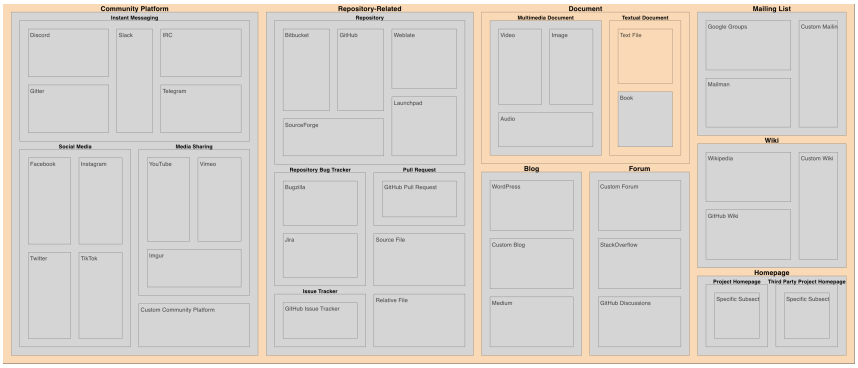
The landscape of scikit-learn starts empty, with only a text file being referenced in the first commit (Figure 3.27a). Just minutes after the first commit (Figure 3.27b), the documentation landscape grows quickly, referencing files across the repository, SourceForge, and some external websites. The SciPy website[54] is mentioned as scikit-learn is built upon it.

SourceForge mentions the Windows installation for the `ctypes` library and, on the following day, the main README file of the project is created for the first time. The README mentions SourceForge, but in this case it is not referencing another project, but a mailing list hosted on SourceForge itself.[55]

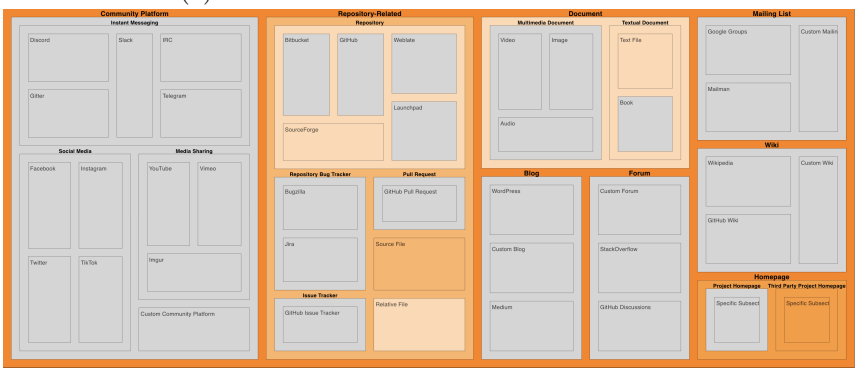
> *Especially when performing evolutionary analysis, the nature of a link that was present in the past can be hard to determine. REs devised on the current snapshot may fail to capture past functionalities of websites, like in the case of SourceForge hosting its own mailing lists.*

Three months later, a *git* mirror of the scikit-learn project[56] is referenced in the README file along with a custom wiki entry[57] about the *git* mirrors for NumPy and SciPy.

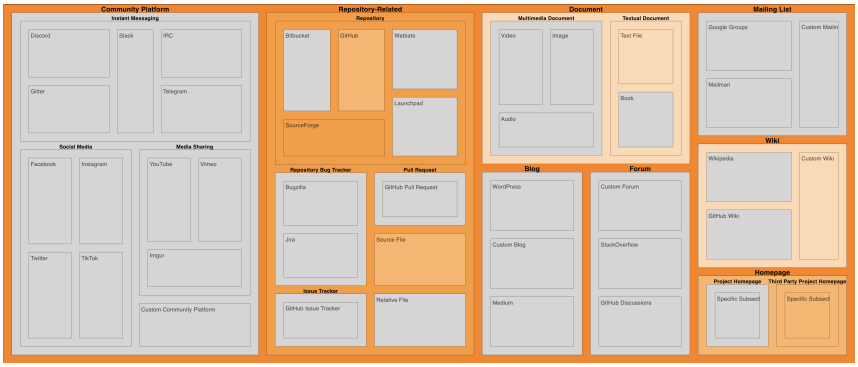
[54] See `https://scipy.org/`
[55] See `https://sourceforge.net/projects/scikit-learn/lists/scikit-learn-general`
[56] See `https://github.com/yarikoptic/scikit-learn`
[57] See `http://projects.scipy.org/numpy/wiki/GitMirror`



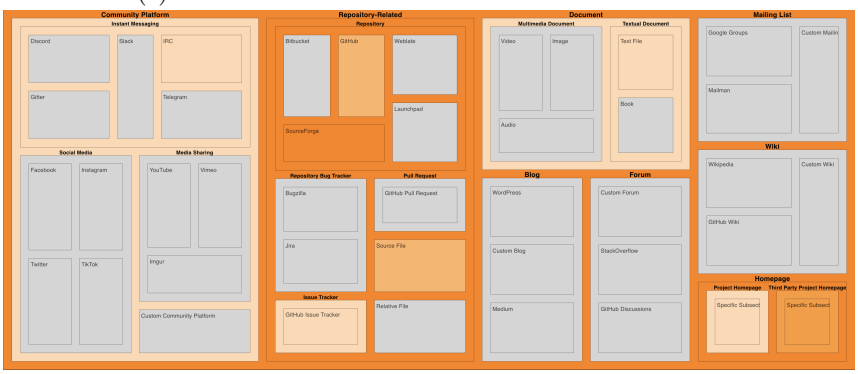
(a) `scikit-learn` on its first commit in 2010.



(b) `scikit-learn` three hours after its first commit.

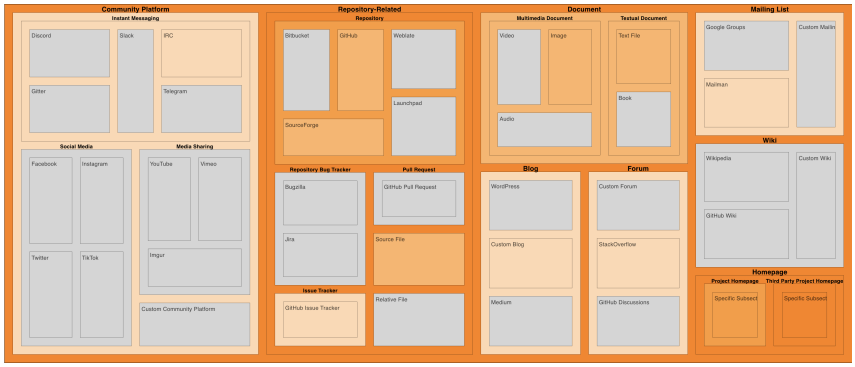


(c) `scikit-learn` three months after its first commit.

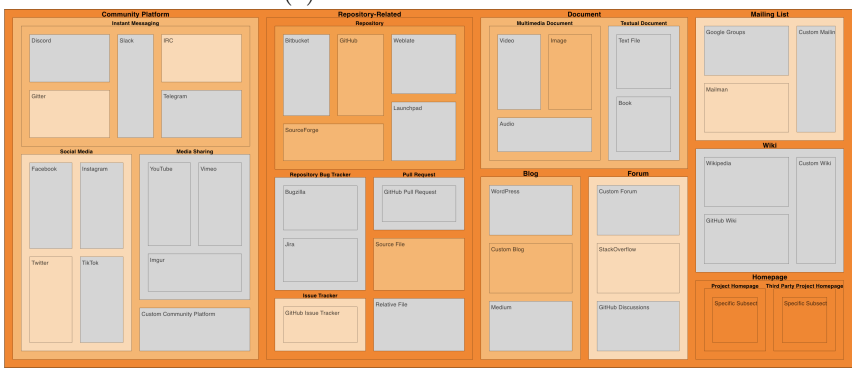


(d) `scikit-learn` in 2011, after one year.

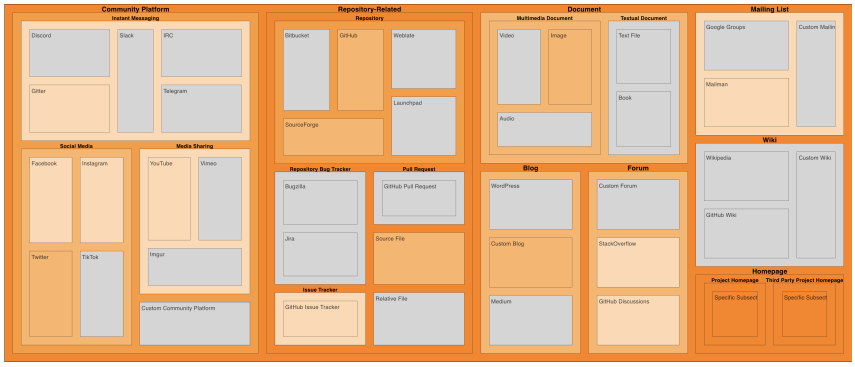
Figure 3.27: Initial documentation landscape of `scikit-learn`. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].



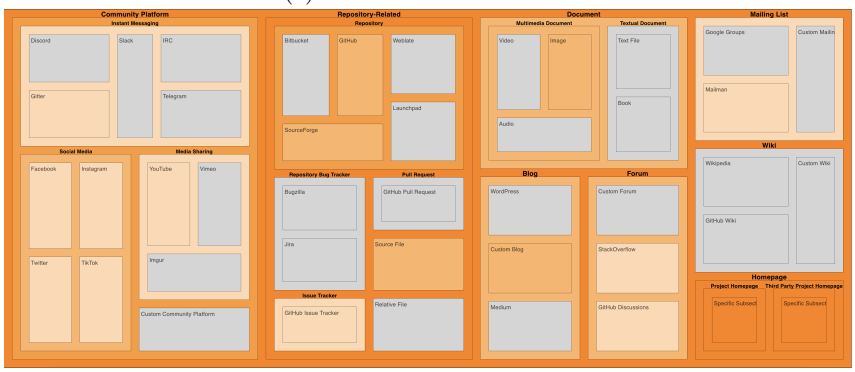
(a) `scikit-learn` in 2016.



(b) `scikit-learn` in 2020.



(c) `scikit-learn` in 2021.



(d) `scikit-learn` in May 2023.

Figure 3.28: The explosion of the documentation landscape of `scikit-learn` between 2020 and 2023. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

Unfortunately, this wiki entry is now unavailable.

> *Evolutionary analysis of the landscape is also impaired by the limited retrievability of the referenced sources. This potential limitation, on the other hand, can be seen as a motivation to actively monitor the landscape, allowing the tooling to leverage the current state of the resources as soon as they appear or are modified in the documentation landscape of a project.*

Both the aforementioned entries will disappear in an update to the README two months later. In June 2010, the `#scikit-learn` IRC channel on `irc.freenode.net` is first mentioned in the README. Within the same short timespan, the README has also now become a `README.rst` in the reStructuredText mark-up syntax. GitHub reappears in the landscape as it is now mentioned for cloning instead of the SourceForge repository of the project. In November, a reference to the homepage of scikit-learn appears for the first time.

In March 2011 (Figure 3.27d), the project's landscape includes more than 30 sources in 8 different categories, also featuring the GitHub Issues tracking system, mentioned in the README for the first time for bug reporting.

From Flowerbeds to the Garden

In August 2012, some images are added to a secondary README file as examples of an included dataset and the landscape stabilizes until 2013, when a custom blog appears. Despite the path looking seemingly secondary: `doc/themes/scikit-learn/static/ML_MAPS_README.txt`, according to the commit message that introduced the link, this blog entry references instructions to edit the README (Figure 3.29). This post gives instructions on how to edit a machine learning cheat sheet of the scikit-learn algorithms, making it relevant to the landscape. The landscape now remains dormant until 2016, when the old mailing list from SourceForge that we were not able to identify via our tool is swapped for a Mailman mailing list. The project had been using a mailing list since it started, and now switched its source. In addition, the README also mentions questions tagged with `scikit-learn` on StackOverflow.[58] The landscape will not change again until 2020 (Figure 3.28a), when the number of source types increases significantly.

File Path: doc/themes/scikit-learn/static/ML_MAPS_README.txt
Past names: None
URL: https://github.com/scikit-learn/scikit-learn/blob/707887d85841f987f1ce82028906a3490456d365/doc/themes/scikit-learn/static/ML_MAPS_README.txt
Number of sources: 5
Number of lines: 94
On date: Tue, 23 Apr 2013 15:16:09 GMT
At commit: 707887d85841f987f1ce82028906a3490456d365
By: Jaques Grobler
Message: add instructions for editing Readme, and script needed for that

Figure 3.29: Commit referencing the blog entry. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

[58] See `http://stackoverflow.com/questions/tagged/scikit-learn`

Welcome to the Jungle

In July 2020, Gitter and Twitter appear in the landscape of the project (Figure 3.28b). One year later IRC is removed from the main README, and GitHub Discussions make their appearance in the Forum category. At the end of 2021, 5 more sources make an appearance all at once. These sources are: The official YouTube channel, a Twitter account for the commits made in the repository, links to accounts on Instagram, LinkedIn,[59] and Facebook. In 2022, a TikTok account is added while, in 2023, the Twitter account dedicated to commits is removed. The documentation landscape of scikit-learn changed significantly in the span of only three years and kept rapidly evolving throughout this period.

> *Regarding the contents of the referenced sources, while some sources are referenced by a project and constitute its software documentation landscape, we do not know to what degree they are used or useful.*

A good example for this is the Instagram page.[60] The page exists, but it can hardly be considered active. The most recent post dates back to June 2023, and is about the release of version 1.3. The page is inactive and with a small number of followers. While it exists in the landscape, it does not contribute much in terms of informative content for the project.

Summary of scikit-learn

Within the span of 13 years, the documentation landscape of scikit-learn has grown steadily and has maintained a stable structure for the first half of its lifespan. From 2020 onwards, the landscape evolves rapidly, covering almost twice as many categories in the taxonomy. Thanks to the views and the exploration performed with RagnaDok, we can see how the landscape evolves at significant snapshots.

The scikit-learn project is peculiar, since many of the sources are in a light shade of orange, indicating the presence of a *single* source per category. The landscape is explicitly curated with a selection of links, one for each category, avoiding to indicate many similar alternatives for the same type of documentation. The exception is the Homepage category, with more than 10 sources, both for the same project and for third-party project links. References to external websites are common and specific sub-sections of the project's homepage saturate the category.

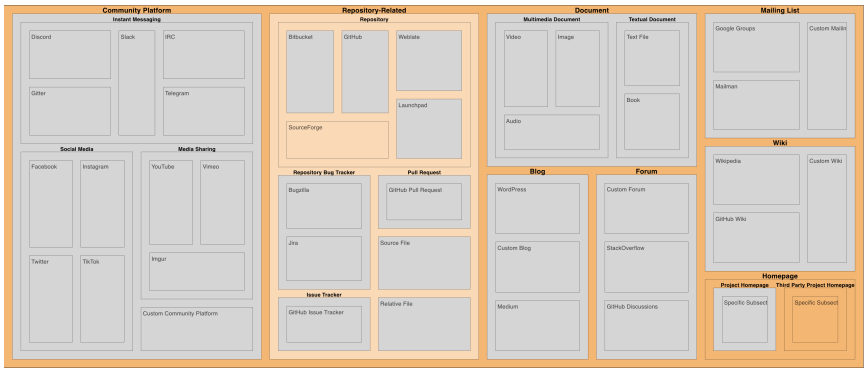
3.6.2 Landscape in a Vase: `fish-shell`

The Fish shell[61] is a modern, user-friendly command-line shell for Unix-like operating systems. It stands out for its interactive features, syntax highlighting, and extensive auto-suggestions that enhance the command-line experience. Fish aims to make the terminal more accessible for both beginners and experienced users. Its powerful scripting capabilities, combined with a vast collection of plugins and extensions, make the Fish shell a popular choice for those seeking a more productive and user-friendly command-line environment [302].

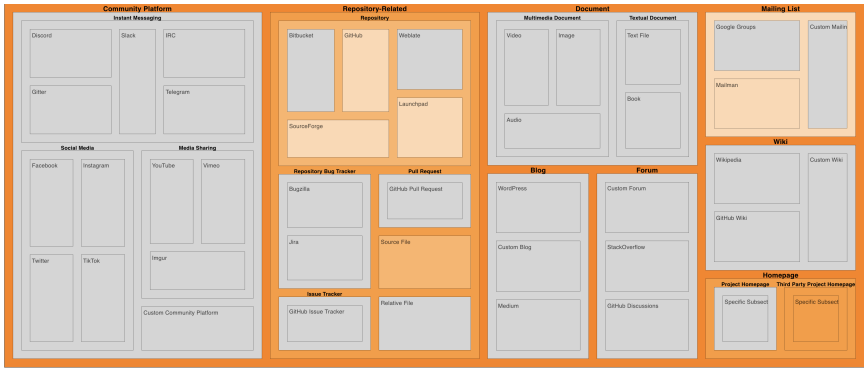
Overview and Evolution

The Fish shell has 17 years of history, with almost 20 thousand commits and shows 10 different README histories. We analyze its landscape without the recovered sources (Figure 3.30).

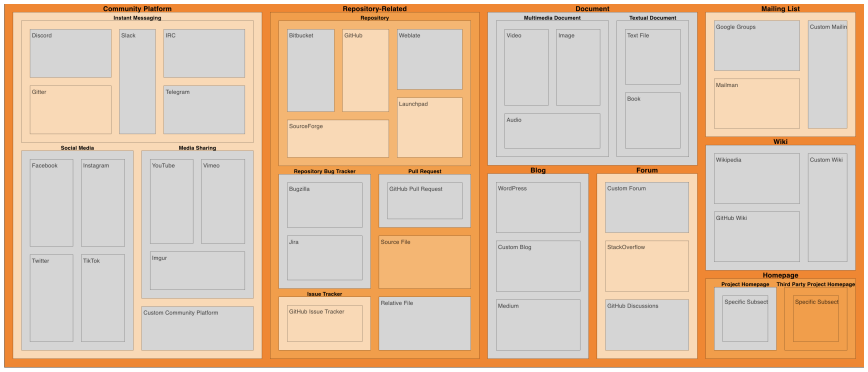
[59] Which remains uncategorized in the taxonomy.
[60] See `https://www.instagram.com/scikitlearnofficial/`
[61] See `https://github.com/fish-shell/fish-shell/`



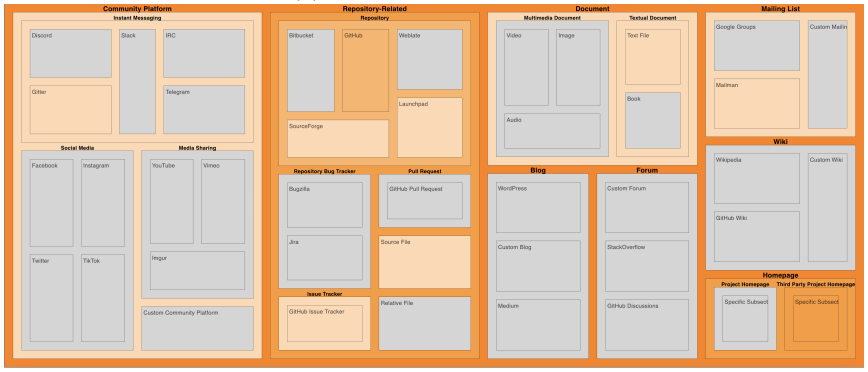
(a) The Fish shell in 2012 when its first sources appeared.



(b) The Fish shell in 2017.

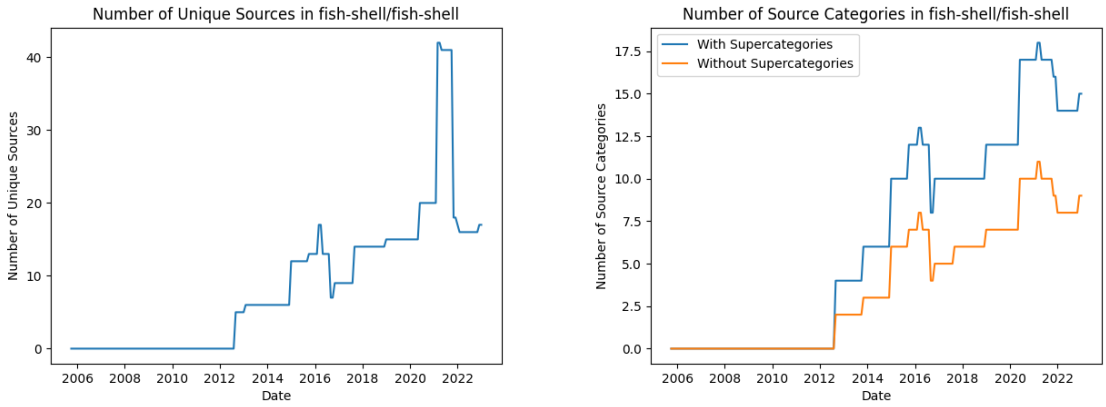


(c) The Fish shell in 2020.



(d) The Fish shell in 2023.

Figure 3.30: Evolution of the documentation landscape of the Fish shell. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].



(a) `fish-shell/fish-shell`: Unique documentation sources over time.

(b) `fish-shell/fish-shell`: Unique source categories over time.

Figure 3.31: Documentation landscape over time of the Fish shell. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

In the software documentation landscape of the Fish shell project, we can see a gradual growth in documentation sources and categories, with two larger spikes in 2016 and 2021 respectively (Figure 3.31). These two spikes match two separate updates to the README of a library used by the project. If we ignore these spikes, as they do not belong to the project itself, we are left with a relatively simple landscape with an equally simple evolution.

The project, created in 2005, features its first readme update only in 2012 (Figure 3.30a). This is because the README,[62] through this seven years gap, has been referencing the official documentation via a command to be executed in the terminal. By running this command, the user can generate the HTML documentation and view it either in a browser or via the help command (similar to a UNIX style `man`). In 2012, SourceForge and a third-party project homepage appear. The SourceForge documentation sources point to a mailing list hosted on SourceForge. The third-party homepage is actually the project's own homepage hosted on a custom domain that does not match the project's name. The Fish shell used to reference its website under a different domain that is now unused.[63]

Soon after, in 2015, we see GitHub being mentioned, both with its wiki and via references to other repositories. We also see Mailman appears in another README of a dependency library.

In 2017 (Figure 3.30b), we find a Launchpad archive in the main README of the repository and the landscape stabilizes for the following three years. In 2020, the project starts referencing GitHub Issues, the official Gitter channel for the Fish shell, and StackOverflow posts tagged `fish` make an appearance (Figure 3.30c). In the final stage of the landscape (Figure 3.30d), we see StackOverflow disappearing again, replaced by `fish` tagged posts on Stack Exchange.

Summary of Fish Shell

The Fish shell project presents a simple landscape where major events in the evolution involve README files of external libraries. We observe how the sources appear and disappear from the top-level README, according to the rise and fall in popularity of the sources we already found in the analysis of the aggregated landscape.

[62] See `https://github.com/fish-shell/fish-shell/blob/149594f974350bb364a76c73b91b1d5ffddaa1fa/README`

[63] See `https://ridiculousfish.com/shell/`

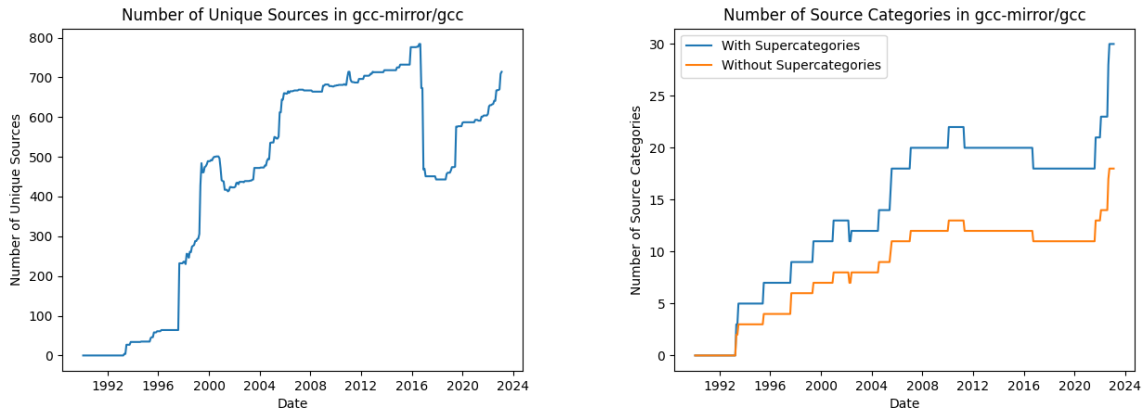
3.6.3 The Secret Gardens: GCC

The GNU Compiler Collection (GCC)[64] stands as a cornerstone within the realm of software development, an open source suite of compilers for multiple languages (*e.g.,* C, C++, Objective-C, Objective-C++, Fortran, Ada, D, Go, Rust) and target hardware architectures (*e.g.,* x8086, ARM, Power ISA). Initiated by Richard Stallman in 1987[65] and developed by the Free Software Foundation (FSF), GCC exemplifies the principles of free and open source software [303].

Overview

Throughout its existence, the project accumulated over 215,000 commits on GitHub and resulted in 320 total separate README histories when analyzed with RagnaDok.

The project presents an empty documentation landscape during its first years of development, until 1993 (see Figure 3.32). The number of categories that the documentation landscape covers has increased gradually, with a spike in 2022–2023 (see Figure 3.32b), while the number of documentation sources peaks early on and then oscillates between 600 and 800 unique sources (Figure 3.32a) with a "clean-up event" that in 2016 reduced the number of unique sources from almost 800 to less than 450.



(a) Unique documentation sources over time for GCC.

(b) Unique source categories over time for GCC.

Figure 3.32: Documentation landscape over time for GCC. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

> *The huge number of sources identified in the landscape of the GCC project is also greatly influenced by the recovered sources [196], although, the number of covered categories remains relatively unaffected.*

The Initial Small Garden

The GCC project contained multiple README files, but without any documentation source, during its first three years of development history. Figure 3.33 shows the evolution of the landscape of GCC between 1993 and 2004.

[64] See `https://github.com/gcc-mirror/gcc`

[65] Formerly GNU C Compiler when started and supporting only the C language.

In 1993 we see the first documentation sources of GCC in the file `gcc/README.ALTOS` (Figure 3.34a). It contains four sources, three e-mail addresses and a source file reference: `jkp@sauna.hut.fi`, `info-gcc@prep.ai.mit.edu`, `jkp@cs.hut.fi`, and `os/exec.c`.

In 1995, a new source in the `gcc/README.FRESCO` file points to `http://www.faslab.com/fresco/HomePage.html`. The page is not available anymore but it looks like the project homepage of the X Consortium's Fresco project (Figure 3.34b).

In 1997, we notice a large spike in the number of documentation sources, with a total of 386. Despite this rapid growth (that is only at its beginning), the documentation landscape presents hardly any new categories (Figure 3.33b). We find various file types, and we see a drastic increase in homepages and source files.

Between the years 2000 and 2004, we can observe a growth in documentation sources (Figure 3.32). With a total of 54 documentation sources in the landscape, we can find a new README: `zlib/README` in 1999 (Figure 3.34c). The file contains 10 sources (31 if we count recovered ones), one of which looks to be the first mailing list that is explicitly mentioned in the landscape (Figure 3.33c). Text file documents are now mentioned as well.

The mentioned "mailing list" points to a now dead URL: `http://web2.airmail.net/markn/articles/zlibtool/zlibtool.htm`. Satisfying the criteria for a custom mailing list, it is categorized as one due to our priority rules for conflict solving (Section 3.3.1).

From an excerpt of the README file:

> *Mark Nelson <markn@tiny.com> wrote an article about zlib for the*
> *Jan. 1997 issue of Dr. Dobb's Journal; a copy of the article is available in*
> *http://web2.airmail.net/markn/articles/zlibtool/zlibtool.htm*

What looked like a mailing list, is in reality a URL that used to point to an article.

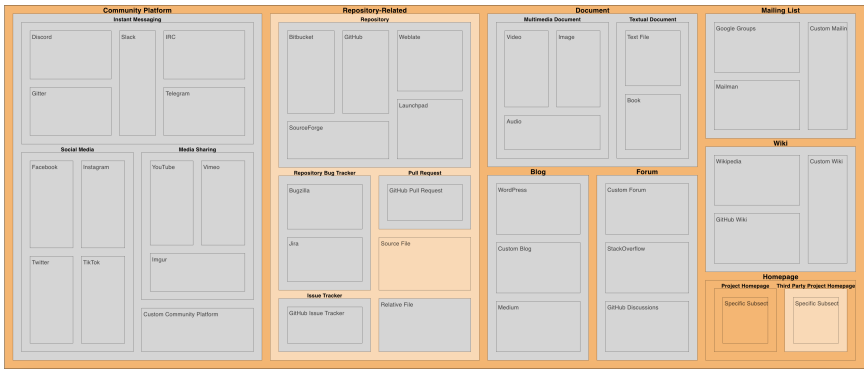
Different types of sources are more or less subject to misclassification depending on how "polarizing" the platforms hosting the documentation are. It is relatively easy to identify GitHub Issues sources, given the standard formatting of the corresponding URLs. Similarly, for mailing lists served by Mailman or other popular applications. Custom platforms and platforms of the past have a higher number of false positives when using REs to capture documentation sources.

Only one year later, we identify the first occurrence of SourceForge. The file `fastjar/README` appears (Figure 3.34d) and quotes its own project on SourceForge, that is still alive to this day.[66]

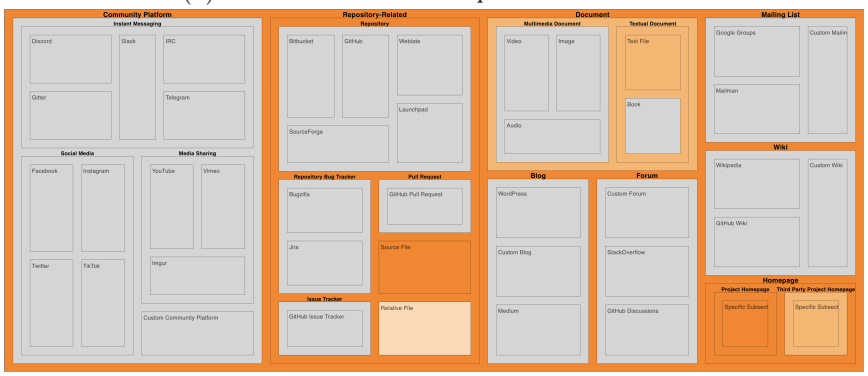
In 2004, we find the first occurrence of Bugzilla in `libjava/README`. The commit message reveals that obsolete information was updated to reference the new bug tracking system in use. From "*Please submit bug reports via this URL:* `http://gcc.gnu.org/cgi-bin/gnatsweb.pl?database=gcc`" it was updated to reference Bugzilla with "*Please submit bug reports via this URL:* `http://gcc.gnu.org/bugzilla`".

We believe that IRC was already present in the landscape of GCC in 2002 at its earliest, despite not being captured by our heuristics (Figure 3.33d), or not being referenced at all in README files. Thanks to the internet archive, we are able to find the earliest version of the IRC node for GCC.[67] The wayback machine shows IRC being present in 2002 and mailing lists already used in 2000.[68] Some of these relevant documentation sources are not linked in README files, they are present only in the main website of the project, one step away from our approach.

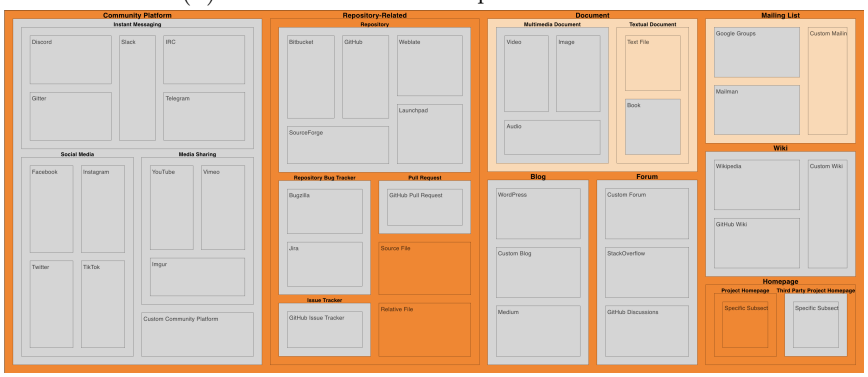
[66] See `https://fastjar.sourceforge.net/`
[67] See `http://web.archive.org/web/20020720151422/irc://irc.oftc.net/`
[68] See `http://web.archive.org/web/20000817013350/https://gcc.gnu.org/lists.html`



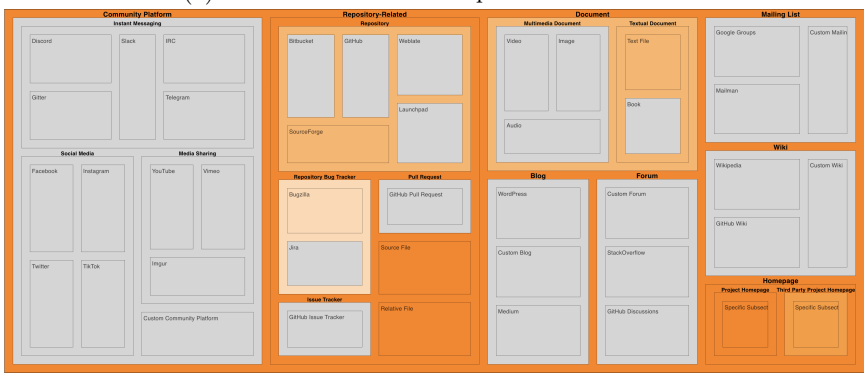
(a) Documentation landscape of GCC in 1993.



(b) Documentation landscape of GCC in 1997.

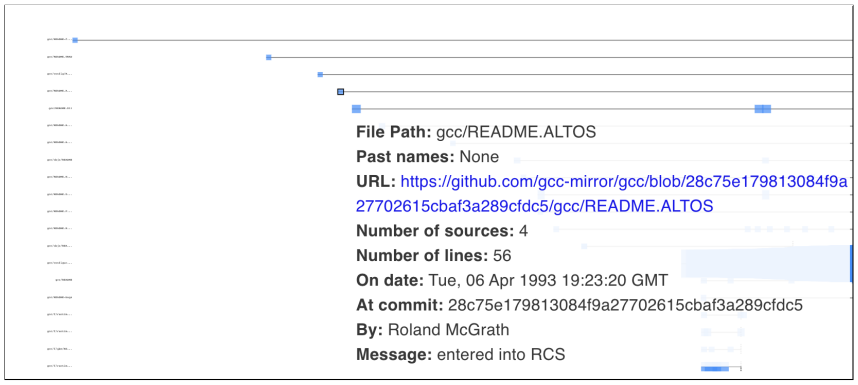


(c) Documentation landscape of GCC in 1999.

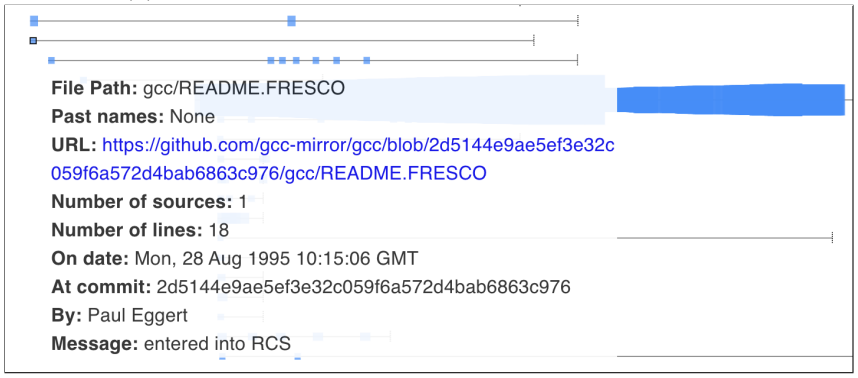


(d) Documentation landscape of GCC in 2004.

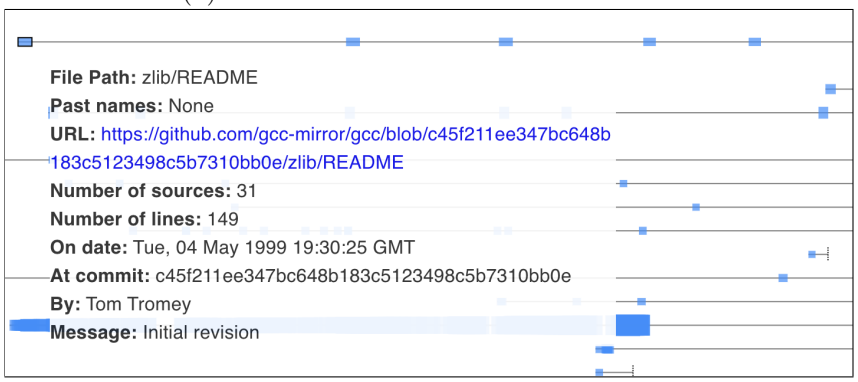
Figure 3.33: Evolution of the documentation landscape of GCC between 1993 and 2004. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].



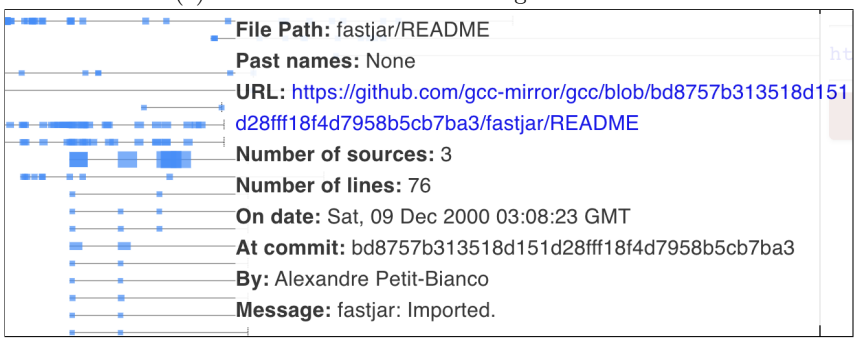
(a) First appearance of documentation sources for GCC.



(b) First real documentation source for GCC.



(c) First occurrence of a Mailing List for GCC.



(d) First occurrence of SourceForge for GCC.

Figure 3.34: README versions with the first appearance of various documentation sources in the landscape of GCC. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

Analyzing the repository is a first step. To reach more documentation sources, the landscape should be extracted recursively from the identified sources. Projects willing to make their landscape discoverable must include a form of curated landscape "bootstrap" (in our case, the README files in the repository). The quality and selectivity of this bootstrap can greatly influence the accuracy of the landscape retrieval.

GCC: Updated Documentation Landscape

The documentation landscape remains stable until 2010, when a custom wiki is referenced in the file `libffi/README`. In 2011, the only occurrence of Bugzilla that existed in the landscape (since it appeared in 2004) is gone, as we can see from the corresponding commit message. The README now points to the generic bug reporting page. In a version of this page from February 21, 2011,[69] the bug reporting page points to the GCC bug database. Bugzilla is still in use at the time but it disappears from the documentation landscape of the project because it is no longer referenced in the repository. Instead, it is indirectly referenced by the official project website.

In 2013, a custom forum appears in `libcilkrts/README`. The source points to the forums for a newly added library.[70] GitHub is referenced for the first time in 2015, in the `libffi/README`, referencing the `libffi` project.[71] The README mentions that details on a merge can be seen at the `git log` of this project. In 2018, the `libcilkrts/README` file is deleted and, with it, the only occurrence of a forum in the documentation landscape so far.

In 2021, we see three new source categories rise, all in the same commit: GitHub Issue Tracker, Custom Forum, and Mailman. The old `libffi/README` has been updated to a new markdown format in `libffi/README.md`, with new documentation sources included in the new version. We see the issue tracker of the `libffi` project,[72] along with two separate mailing lists: `https://sourceware.org/mailman/listinfo/libffi-announce` and `https://sourceware.org/mailman/listinfo/libffi-discuss`. The second was erroneously categorized as a forum.[73]

Analyzing the different levels of the URL path (similarly to the prioritization of category specificity), could improve the source extractor's accuracy.

Summary of GCC

GCC's history spans three decades, covering a rather small slice of the taxonomy, with substantial entries appearing only after 2000, almost a decade after the project's first commits. Similarly to the first case study (scikit-learn, see Section 3.6.1), the documentation landscape has a "mini-explosion" with new documentation sources in recent years. Older projects, such as GCC and the Apache projects, come from an era where the main resources were listed in the project's website. Sources are organized and mentioned there, rather than in a README file.

Moreover, many of the sources we mentioned come from README files of libraries, not belonging to the main project itself. In fact, originally, the GCC project already had about 60 pages of written documentation when it was first released. This documentation is contained in many `.texi` files[74] that must be processed in order to make them readable. The documentation landscape of GCC has always existed, but it was a secret garden, not easily reachable via README files nor traceable with our approach based on RagnaDok [196].

[69] See `http://web.archive.org/web/20110221073604/http://gcc.gnu.org:80/bugs/`
[70] See `http://software.intel.com/en-us/forums/intel-cilk-plus/`
[71] See `http://github.com/atgreen/libffi`
[72] See `https://github.com/libffi/libffi/issues`
[73] Due to presence of "`discuss`" in the URL (see Appendix B).
[74] See `https://github.com/gcc-mirror/gcc/tree/master/gcc/doc`

The chaining and retrieving the readme files in the repository allows to focus on all the potential sources present in the landscape of a project, even those introduced by dependencies and not directly related to the project. This introduces noise and bloats the landscape, sometimes overshadowing the most relevant sources.

3.7 Summary

We presented a preliminary study of the evolution of the landscape in the last two decades. We showed how the aggregate documentation landscape can provide insights on the tools developers use to create and share software documentation. We presented preliminary data on the shift of emphasis towards communication platforms in collaborative development (more in Section 4.7).

The case studies highlight how many different shapes we can find for the documentation landscape of a software system. Depending on the system and throughout its history, we can identify different health conditions of the landscape snapshots, also related to the ease of discoverability of the sources and the reachability of their contents.

This initial overview highlights how the documentation landscape has been constantly growing, both in size and complexity (number and variety of categories and sub-categories). One of the main contributions of the presented study is the final taxonomy of the current documentation landscape (Figure 3.5). In the wake of the constant reshaping affecting the landscape, the health status of some sources becomes even more critical. As we will see in the following chapters, instant messaging applications used as a documentation means introduce a number of challenges for the generated documentation (*e.g.,* indexing, persistence). Keeping track of the landscape's evolution to identify weak links in the current practices is fundamental to promptly identify problems and potentially relevant research topics about modern software documentation.

4

Modern Software Documentation

In the new landscape outlined in the previous chapter, classical software documentation, as it was conceived and intended decades ago, is not the only reality anymore. Official documentation from authoritative and official sources is being replaced by real-time collaborative platforms and ecosystems that have seen a surge, influenced by changes in society, technology, and best practices. These modern tools influence the way developers document the conception, design, and implementation of software. As a by-product of these shifts, developers are changing their way of communicating about software. Where once official documentation stood as the only truth about a project, we now find a multitude of volatile and heterogeneous documentation sources, forming a complex and ever-changing documentation landscape.

Software projects often include a top-level readme file with important information, which we leverage to identify their software documentation landscape. Starting from ∼12k GitHub repositories, we mine their readme files to extract links to additional documentation sources. We present a qualitative analysis, revealing multiple dimensions of the documentation landscape (*e.g.,* content type, source type), highlighting important insights. By analyzing instant messaging application links (*e.g.,* Gitter, Slack, Discord) in the histories of readme files, we show how this part of the landscape has grown and evolved in the last decade.

We show how modern documentation encompasses communication platforms, which are exploding in popularity among developers. This is not a passing phenomenon: On the contrary, it entails a number of unknowns and socio-technical problems the research community is currently ill-prepared to tackle.

4.1 From Documentation to Communication

Times are changing. This is even more true for software engineering. Major shifts have occurred, induced by the emergence of platforms like GitHub and StackOverflow, fundamentally changing how developers (and users) communicate about software projects: Mailing lists and forums are declining in favor of multi-media instant messaging platforms, such as Gitter, Slack, Discord, and GitHub Discussions, *e.g.,* [43, 66, 95, 113, 143, 144, 174, 175, 181, 183, 209, 225, 241, 244].

Software documentation, a critical asset for developers [5], has been studied extensively with respect to its quality and usefulness [6, 45, 54, 79, 83, 192, 231, 265]. Nevertheless, the impact of the subtle but constant drift induced by new platforms is still to be evaluated. What are the implications for program comprehension if a tweet can influence how developers treat a bug [158]? Can the tweets on the usage of an API also serve as documentation? Classical software documentation, as we have known it, is being replaced by "communication".

Documentation went from a clunky, and rather unloved, endeavor to becoming a fast-paced and volatile side dish. The utopia of *"on-demand documentation"* by Robillard *et al.* [193], is being replaced by a dystopia of an ever-changing landscape; documentation is waved away with sentences like *"check Discord"* or *"it's in the pull request comments."*

This change is more than just cosmetic, it is considerably affected by the richness of new media, influencing the cognitive processes that underlie communication [190]. Modern media-rich platforms offer vastly different mechanisms which are simply not there in classical electronic communication means. Moreover, developers do not only hold ephemeral discussions that they must be able to access now. They share knowledge (*e.g.,* code examples, screenshots, how-tos) that is important for them in the future, and they do not have (or rather: take) the time to persist it in a classical software documentation form (*e.g.,* APIs, Wikis). Instant messaging is just too enticing for that. But, developers will still need long-term access to this knowledge and they want to keep it searchable [304][1] and organizable[2] [305]. They choose their platforms accordingly, for example, avoiding limitations in retrievable history [306], and are willing to pay significant sums for such services [307].

As the cards on documentation are being reshuffled, things seem rather murky: What happens to the body of knowledge contained in the repositories of classical communication platforms? What is the impact on standard software documentation? How do developers use modern platforms, and what does this imply for software documentation practices?

The Spectrum Example

*Spectrum, a multi-forum community hosting platform, was hosting dozens of software related communities about frameworks (*e.g., *React, Laravel), UI design (*e.g., *Figma), front-end coding (*e.g., *CodePen), and developers' networks in general (*e.g., *SpecFM). On Aug 24, 2021, to preserve history while pushing forward the adoption of new communication infrastructures, it was announced that "the time has come for the planned archival of Spectrum to focus our efforts on GitHub Discussions" [308]. Spectrum has become "read-only – no* `404`*s or lost internet history." The Spectrum team acknowledged the importance of conversations held on the platform and tried to avoid the limitations of relying on the Internet Archive for preservation [250]. Many Spectrum communities had already moved to GitHub Discussions, for reliability and flexibility reasons: Having code and the community in the same place outweighed other factors in the decision to change.*

We present an overview of the *documentation landscape* exploring beyond current trends in documentation platforms, focusing on the relationship between documentation and communication platforms, exemplified by the tendency in a project's readme to include the latter as an indirect source of the former.

We show the most representative values in different dimensions characterizing the landscape. We then proceed more in-depth with the history of modern communication platforms. We show how some platforms have seen increasing adoption, reached a plateau, and finally started their decline. Our analysis provides insights into the many implications of this ongoing phenomenon for software documentation. Finally, we discuss possible features that future platforms should have to mitigate some of the perils introduced by these continuous shifts.

[1] Especially for large communities, without limitations, as reported in this blog post.
[2] As demonstrated by the presence of an ecosystem built on top of instant messaging applications.

4.2 Mapping the Documentation Landscape

The paradigm shift affecting modern software documentation influences critical aspects of development and management of software projects. In the literature, there is a lack of systematic approaches to classification and integration of multiple documentation sources. Mapping and reifying the documentation landscape is fundamental to allow automatic discoverability of a system's documentation. Our approach aims at extracting, integrating, and analyzing attributes and informative content of relevant documentation types. Fast communication through instant messaging applications is a form of unstructured documentation. We use this source as a case study to highlight its specific criticalities.

4.2.1 Definition and Taxonomy

The *documentation landscape* of a software system amounts to all the possible sources of information pertaining to the design, implementation, program comprehension, maintenance, and evolution of a software project. In an initial manual exploration of the possible documentation types linked in GitHub repositories, we identified thirteen possible sources and classified them according to their nature – the *archetype* – and to different metrics for each archetype (Figure 4.1) [180].

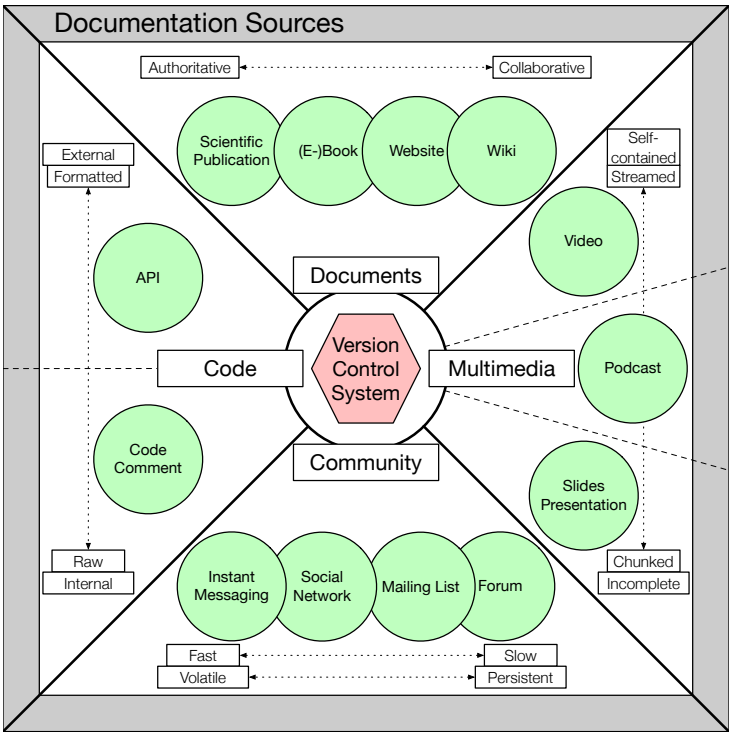


Figure 4.1: Documentation landscape of a software system.

There are four archetypes: **Documents**, **Code**, **Multimedia**, and **Community**. These surround the central point of source code in a version control system. For sources in each archetype, metrics are spanning the horizontal or vertical axis. For example, in the *Community* archetype there are slower (*e.g.,* mailing lists) and faster (*e.g.,* instant messaging) sources, volatile and persistent ones. Each source has multiple possible instances. For example, many instant messaging applications are possible documentation sources (*e.g.,* Gitter, Discord, Slack).

The taxonomy in Figure 4.1 abstracts specificities of the sources found in actual repositories. With respect to the preliminary analysis presented in Chapter 3, this manual taxonomy highlights some of the differences between the identified source types. Some sources are more authoritative, with contents that has been checked by multiple stakeholders (*e.g.,* peer review in scientific publications), while others are unstructured, fast, and volatile. In this taxonomy, we also recover some of the sources that did not emerge from the preliminary study, as they are so imminent to the repository that no mention of them could be found explicitly (*e.g.,* code comments).

Our definition of documentation landscape is encompassing multiple source types and concrete instances for each source. In the next section, we explore how these sources can be retrieved and characterized according to different dimensions. We start from the most prominent and easily accessible entry point to the landscape of a project: Its *readme* file.

4.3 A Dataset of Sources with DwarvenMail

This section details the procedure and tool support (DwarvenMail) we implemented to collect the data for our analyses (Figure 4.2). We present the initial dataset, the mining procedure, our manual annotation, and details about the individual analyses we performed.

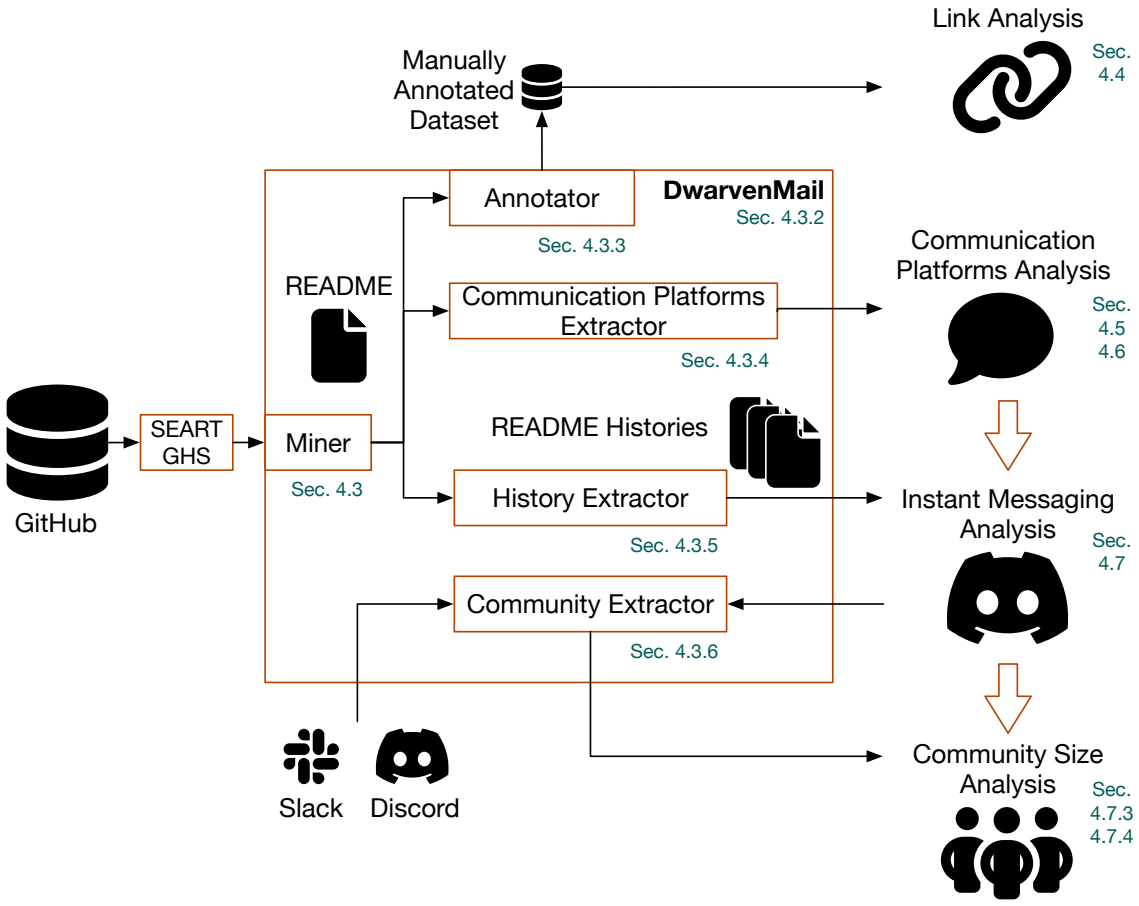


Figure 4.2: DwarvenMail and analyses overview.

4.3.1 Project Mining

Starting from all repositories currently hosted on GitHub we used SEART-GHS [52] to compose a relevant dataset, applying the following filtering criteria: at least 2,000 commits (*i.e.,* to eliminate toy projects), more than 10 contributors (*i.e.,* to ensure that a certain number of people need to interact with each other to tackle the development effort), and more than 100 stars (*i.e.,* to ensure that the projects are relevant to at least a handful of people). We only considered projects created before July 1, 2022 and excluded forks [309]. SEART-GHS currently monitors about 1.2M GitHub repositories. Projects excluded in SEART-GHS for having less than 10 stars [52] would have been excluded by the more restrictive criterion we applied, removing projects with less than 100 stars. We performed the filtering on July 14, 2022, resulting in 12,461 projects exported as JSON input for DwarvenMail. After removing 374 aliases and 6 renamed forks, the final scraped dataset consists of 12,081 projects.

*Filtering based on the number of stars might not be sufficient to select relevant projects. Nevertheless, starring can be important for project developers and managers [35]. We used this criterion as a common method for filtering out toy projects in GitHub (*e.g., *see [269]).*

Table 4.1 presents an overview of the projects according to their languages.

Table 4.1: Projects and represented languages.

Language	Projects						
	All	**CP**	**CP %**	**Δ_{CP}%**	**IM**	**IM %**	**Δ_{IM}%**
C	1,240	548	44.2%	−13.1%	248	20.0%	−9.2%
C#	543	378	69.6%	+12.3%	214	39.4%	+10.2%
C++	1,707	926	54.2%	−3.1%	469	27.5%	−1.7%
Go	677	443	65.4%	+8.1%	276	40.8%	+11.6%
Java	1,510	860	57.0%	−0.4%	432	28.6%	−0.6%
JavaScript	1,528	899	58.8%	+1.5%	440	28.8%	−0.4%
PHP	733	379	51.7%	−5.6%	165	22.5%	−6.7%
Python	1,806	1,094	60.6%	+3.3%	557	30.8%	+1.7%
Ruby	406	238	58.6%	+1.3%	103	25.4%	−3.8%
Rust	244	163	66.8%	+9.5%	112	45.9%	+16.7%
Shell	205	93	45.4%	−11.9%	35	17.1%	−12.1%
TypeScript	895	575	64.2%	+6.9%	314	35.1%	+5.9%
Other/Unspecified	587	328	55.9%	−1.4%	160	27.3%	−1.9%
Total	**12,081**	**6,924**	**57.3%**		**3,525**	**29.2%**	

The *All* column shows the total number of projects, the *CP* column the projects where we could identify communication platforms (see Section 4.3.4), and the *IM* column the projects with instant messaging platforms. Percentages are derived with respect to the *All* column, while Δ percentages are relative to the *Total* row percentages.

Overall, 57.3% of the projects we analyzed feature at least one communication platform. An interesting observation is that systems written in "lower level / traditional" languages (C, PHP, Shell) tend to be below the overall average, while systems written in more "modern" languages (C#, Go, Rust, TypeScript) are more inclined to feature communication platforms.

The difference is even more evident for recently popularized languages if we consider projects with instant messaging platforms (*e.g.,* Go and Rust increase from +8.1% to +11.6% and from +9.5% to +16.7% respectively).

We performed multiple *One Proportion Z-Tests* (one for each language) and the difference in the proportion of projects using communication platforms for each language (r) and the overall dataset (r_0) is statistically significant for C, C#, Go, PHP, Rust, Shell, and TypeScript ($H_0 : r = r_0$, two-tailed Bonferroni corrected p-value < 0.0038). The same results hold for projects with instant messaging platforms.

4.3.2 Tool Support: DwarvenMail

To support our analyses, we developed DwarvenMail, a Python application to scrape GitHub and extract information about projects' readme files and their history. It features an object-oriented domain model to facilitate the extraction of insights from exploration.

DwarvenMail also supports manual inspection, link extraction, and classification from readme files (see Section 4.3.3). DwarvenMail takes the list of projects in the dataset and uses the REST API of GitHub and web scraping [310] to extract the information needed to build its internal domain model. It is implemented as a multiprocess application to speed up the scraping. Each process uses a different API key to access GitHub in parallel through PyGitHub [311]. Parallelization is handled at project level: Each process gets a project from a queue and starts to fetch the data. Processes are also responsible for not exceeding GitHub rate limits.

4.3.3 Manual Annotation

To examine the documentation sources and communication platforms of the projects, we performed a qualitative analysis of their README files. We relied on open card sorting, a well-established method for knowledge elicitation and classification [17, 166, 232, 261, 262], to incrementally refine the list of possible sources with flexible categories.

We manually reviewed the README files of the projects, extracted their links to documentation sources and organized them into categories. Given the considerable effort needed to annotate readme files manually, we opted for a saturation approach [212]. We started with a sample set of 35 projects selected through stratified sampling, ensuring a balanced distribution among programming languages.

Two authors independently annotated each project README. Then we repeated the process in subsequent batches with 5 projects per batch until no new labels were added in two consecutive batches. We reached saturation after annotating 60 projects. In the end, we discussed conflicts and merged categories where needed. The process resulted in 2,349 links with 282 link types, which we discuss in Section 4.4. The creation of manually annotated datasets was supported by the *Annotator* module of DwarvenMail (Figure 4.3).

An annotator ran the Flask application locally, pulled from *git* the latest updates by other annotators, started a batch of annotations, committed, and pushed the modified files.

The Annotator's homepage shows a list of projects to annotate and the annotation status (*i.e.,* who annotated what). Selecting a project opens the project annotation page (Figure 4.3) where one can browse the readme of the selected project.

The project annotation page uses two side-by-side panes to present the readme. The left one represents the raw Markdown version of the readme. The right pane shows a partially rendered version (*i.e.,* similar to what a user sees on GitHub).

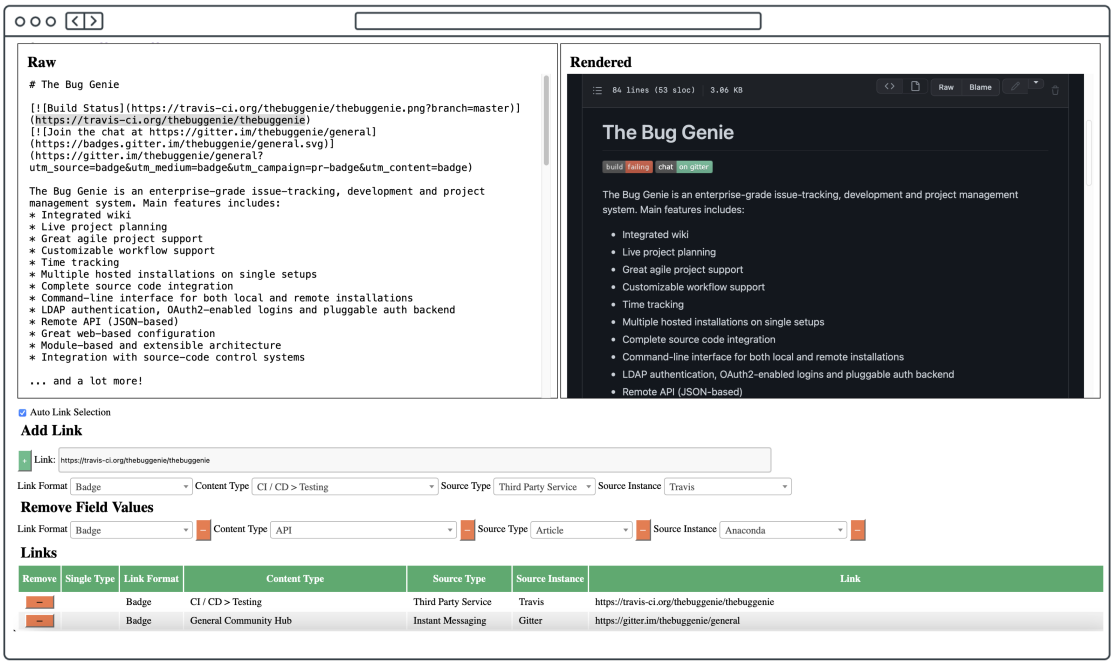


Figure 4.3: DwarvenMail Annotator – Project annotation page.

4.3.4 Parsing Links: Strategy & Heuristics

We performed a quantitative analysis of communication platforms in readme files (Sections 4.5 and 4.6). To support automatic platform extraction in such a large number of projects we used an approach based on Regular Expressions (RE).

For each communication platform that we discovered, we devised REs that would match the link as closely as possible, while retaining sufficient generality to abstract specific aspects (*e.g.,* project name, internet domain, optional protocol). Possibly more than one RE has been associated with each platform. To fine-tune the REs, DwarvenMail features a detailed log generation for manual inspection of candidate and invalid links during the refinement.

DwarvenMail parses all the links in a readme according to the set of identified REs. When a link is found, specific exclusion criteria are applied. A set of rules removes links to images, badge icons, platforms' generic homepages, and partial or invalid URLs (*e.g.,* shorthands for Markdown sections captured by the REs).

The remaining links are normalized in a standard format and duplicates are removed. Platforms not directly linked in the readme (*e.g.,* collected in a list on the *Community* page of the project website) are omitted by the employed scraping algorithm. To reduce the false positive rate, DwarvenMail also verifies that links point to valid web pages (*i.e.,* the server does not respond with an `HTTP 404 Not Found`). After this refinement, we obtain the final set of communication platforms referenced by the project readmes.

4.3.5 Parsing readme Histories

For projects referencing Gitter, Slack, and Discord as communication platforms, we analyzed the history of their readmes to discover when those platforms appeared for the first time. In this case, the approach outlined above to exclude invalid links (Section 4.3.4) would not produce the desired results, because a link that is not valid today could have been valid in the past.

This cannot be checked without an archive, or historical information. Hence, in our approach we assume that links with proper format were valid in the past. To reduce false positives, we used the most specific format able to capture the link.

4.3.6 Community Size

We include the Discord and Slack community size (*i.e.,* number of members) in our domain model. The most popular way to add people to a Discord server is through an *invite link* [312]. Clicking on an invite link, brings the user to a page with metadata about the server (*e.g.,* number of total members, number of online members). We gathered Discord community sizes by scraping the data from these invite pages. In the case of Slack, only 15% of the projects in our dataset have information about the community size on the invite page.

Extending the percentage of projects whose community size is correctly scraped could improve the reliability of results discussed in Sections 4.7.3 and 4.7.4.

> *More effort is needed to explore communities that do not conform to a standard and/or customize their invite link to pursue specific goals (*e.g., *authorization workflow, authentication, spam prevention, analytics).*

4.3.7 Data Availability and Replication Package

We provide a replication package, publicly available on Figshare [313], containing the source code of DwarvenMail, the input dataset, the manually annotated projects, the serialized domain model of the scraped dataset, charts and tables exported from DwarvenMail.

4.4 Documentation Landscape

We define the documentation landscape of a software system as all the possible sources of information able to support design, implementation, comprehension, maintenance, and evolution of the system. Software documentation is a fundamental asset for developers and practitioners [5], when it is correct and up-to-date [45, 54, 79, 83, 192, 231], with its costs and benefits [265]. Modern software documentation is an ever expanding field. New sources include blogs [170], Twitter [251], StackOverflow [176], instant messaging applications [43, 66, 113, 144, 174, 175, 181, 183, 209, 225, 241, 244], news aggregators [10], and forums [95].

GitHub readme files in Markdown (`.md`) format are a good starting point for a project from where all relevant documentation should be reachable. Documentation sources in readme files can either be directly referred to or behind multiple steps of indirection. An example of the former case is an invitation link that can be copy/pasted directly in Discord to access the community server of the project. In the latter case, the readme could point to a community web page which in turn contains links to the mailing list, a Slack channel for Q&A, and potentially other communication and documentation sources.

The manual annotation presented in Section 4.3.3 produced 282 *single type* link tags. The links can come in many flavors thanks to the markdown format, from pure textual hyperlinks to badges and images that link external resources. We inspected them and identified three key dimensions of the documentation landscape: *content type, source type,* and *source instance*. We split single type tags into these three dimensions. We analyzed examples of each link type to disambiguate or enrich the classification when the original annotation had missing information.

Table 4.2 shows the top-15 most representative values for each dimension. The complete list of tags is available in the replication package [313].

Table 4.2: Top-15 most relevant tags, number of projects, and links for each dimension. The percentage indicates the ratio of projects containing at least one link with the specified tag.

(a) Content type.

Projects	Content Type	Links
36 (60%)	General Community Hub	141
29 (48%)	Official Documentation	97
28 (47%)	License	68
25 (42%)	Contributing	56
23 (38%)	Issues	52
23 (38%)	CI/CD	50
21 (35%)	Project Repository	76
20 (33%)	Relevant Projects	132
20 (33%)	Dependency/Environment	84
19 (32%)	Releases	60
16 (27%)	In-Repository Resource	67
16 (27%)	Package Repository	47
14 (23%)	CI/CD → Testing	24
13 (22%)	Installation Instructions	24
11 (18%)	Code Coverage	22

(b) Source type.

Projects	Source Type	Links
55 (92%)	Homepage/Website	436
41 (68%)	Collaborative Platform	188
36 (60%)	Third Party Service	169
34 (57%)	Wiki	125
32 (53%)	Repository	166
28 (47%)	Source-file/Source-folder	151
25 (42%)	Instant Messaging	84
11 (18%)	Auxiliary README	21
10 (17%)	Readme Section/Anchor	44
9 (15%)	Mailing List	27
9 (15%)	Forum	20
8 (13%)	Image/GIF	21
8 (13%)	Blog	20
7 (12%)	Email Address	12
6 (10%)	Video	13

(c) Source instance.

Projects	Source Instance	Links
33 (55%)	GitHub	179
16 (27%)	GitHub Workflows	50
13 (22%)	GitHub Releases	27
12 (20%)	Travis	22
11 (18%)	Gitter	34
9 (15%)	Google Groups	24
7 (12%)	Discord	18
7 (12%)	Codecov	14
7 (12%)	Python Package Index	13
6 (10%)	Twitter	12
6 (10%)	StackOverflow	9
5 (8%)	Slack	18
5 (8%)	Maven	11
5 (8%)	Read the Docs	5
4 (7%)	GitHub Profile	108

The three key dimensions we propose to describe the documentation landscape of a software system are content type, source type, and source instance, exemplified as links in GitHub readmes.

Source type, source instance, and content type could describe a link like: "This link is in the form of a *Badge*, it points to a *Wiki* on *Travis.com*, and contains information related to *CI / CD*." Each dimension is instantiated with one of the possible tags for that category, forming a signature of the documentation source pointed by the link.

Exploring the identified key dimensions and their interplay could improve the automatic extraction of links and their features, to characterize and understand the (evolution of the) documentation landscape.

Link format: Link formats come in many flavors, also due to the fact that markdown files, while being textual, are usually inspected using a multimedia capable web browser. Badges, for example, are very common in GitHub readme files, used to convey imminent information through iconic representation of a summary of the pointed resource (*e.g.,* build status passing) where the link itself allows, if followed, to reach more extensive information (*e.g.,* build process report). Masked links are another common practice to add links to markdown documents.

Not all links in a raw readme file are human readable links in the rendered readme.

Content type: This is the primary dimension of the documentation landscape, denoting **what** kind of information is present in the landscape. There is a smooth gradient in content types regarding the number of links, but it is worth noting that there is no "standard", but rather project-specific landscapes. Most relevant are *general community hubs*: Discord servers, Slack workspaces, Gitter rooms, IRC channels, mailing lists, and forums, with their internal structure for different topics, dedicated to a general community of users and practitioners.

*Content type is relevant to interpret **what** a piece of documentation is about. There is no standard to the documentation landscape, each project develops its own. Even the top content types (community hubs, official documentation) are present in only half of the projects.*

Source type: The *source type* dimension refers to the format of the content at the link's destination. This dimension is relevant for automatically extracting the documentation landscape since it determines how the content can be retrieved and parsed. *Homepage / websites*, the most relevant source type by a large margin, can be scraped with traditional web scraping techniques. *Collaborative platforms* like GitHub and Bugzilla could be addressed via their custom APIs. *Image / GIF* → *Screenshots*, further down in terms of relevance, would benefit from image segmentation and analysis approaches to extract, for example, documented user interface features. We also notice that links to mailing lists are fewer than those to IM applications, a trend we analyze in more detail in Section 4.5.

*Source type captures **how** documentation is presented and how it can be accessed. Almost all projects feature a head quarters website, i.e., the go-to place to learn about a project. These starting points are then often complemented by a plethora of other sources, ranging from Wikis to forums and instant messaging platforms.*

When analyzing the evolution through time of a readme file we detect in many cases that the source types come and go, inducing "tectonic movements" in the landscape, as we can observe in the example depicted in Figure 4.4.

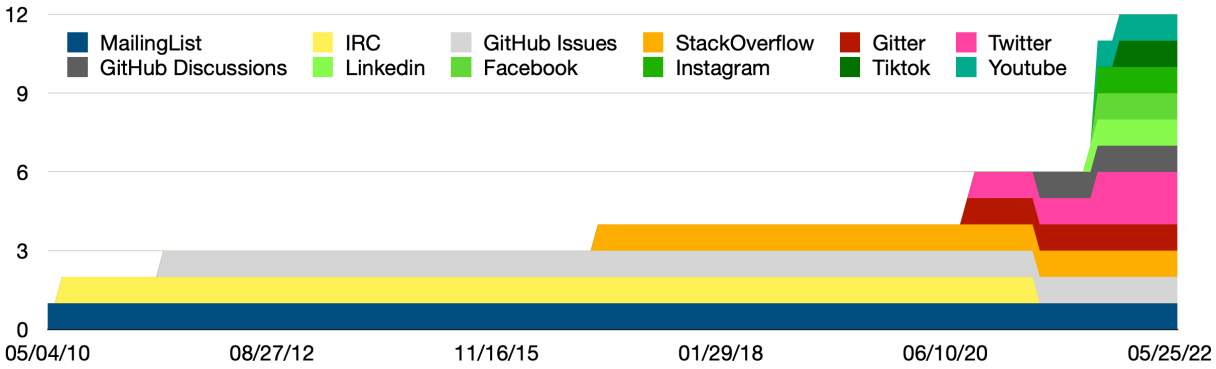


Figure 4.4: Evolution of communication platforms in the `scikit-learn` project.

The Scikit-learn project, born in 2010, sees a mailing list as its initial documentation landscape, complemented shortly after by an IRC channel (which stopped existing a decade later). GitHub Issues is added within the project's first year, while StackOverflow becomes part of the landscape in 2017. It is within the past 2 years that the landscape experiences an earthquake, with many new sources appearing, while the IRC channel is removed (it is worth noting that IRC and its successor, Gitter, co-exist for a year). At the time of writing the project in question features 11 different sources.

The documentation landscape of projects evolves together with the project. Especially in the past few years the source types have exploded in number, rendering the landscape highly dispersive.

The fact that there are more sources does not imply that the overall documentation of the system is better, on the contrary: We have observed an overall trend toward more volatile sources, mostly due to the rise of instant multimedia messaging platforms.

Source instance: The third dimension is a derivate of source type. For each type we can have multiple possible source instances, usually of a competing nature (see Section 4.6) with a similar purpose. Rather unsurprisingly for GitHub projects, GitHub itself with related instances of profiles, workflows, releases, and instant messaging (*i.e.,* Gitter) takes top three, the 5th, and the 15th places. Services for package repositories (*e.g.,* Python Package Index [314], Maven [315]) and Continuous Integration / Continuous Deployment (CI/CD), like, for example, Travis CI [316] and Codecov [317], for messaging applications, like Slack [318] and Discord [319], and also articles on the Medium platform [320] represent interesting research avenues.

*Source instance can be seen as **where** (or by whom) documentation is "hosted."*

Source instances vary wildly, and new players constantly enter the stage. For example, recent changes in the pricing model of Slack might have influenced the ongoing mass migration toward other instant messaging platforms, of which there are dozens, with Discord quickly becoming the preferred alternative.

Tags in the three dimensions appear in different combinations, not all equally likely. Further research on the most common patterns could shed light on form and content interplay in software documentation.

4.5 Modern Communication Platforms

One of the recent major shifts in software development has been the emergence of various multimedia instant messaging platforms, such as Slack [318], Discord [319], and Gitter [321].

They not only experienced an increase in popularity but also seem to be a major suspect for the decline of other classical communication means, such as mailing lists and forums. We start by analyzing the platforms actually used by projects in our dataset.

The scraping, based on regular expressions (see Section 4.3.4), took place between Aug 28 2022 at 21:01 and Aug 30 2022 at 02:12, leading to 12,081 scraped projects. Of those, 6,924 (57.3%) mention at least one such modern communication platform in their readme files: 2,897 had 1 platform link, while 4,027 had 2 or more platform links. The remaining 5,157 projects had no platform links. The percentage is higher than the one reported by Käfer *et al.* [119] (57.3% vs. 46.7%), which can be explained by the fact that their analysis dates back 4 years.

We grouped communication platforms into three main categories: asynchronous, instant messaging, and social media. Figure 4.5 summarizes the number of links in readmes (*Links*) and the number of projects with at least one link (*Projects*) for each type of platform.

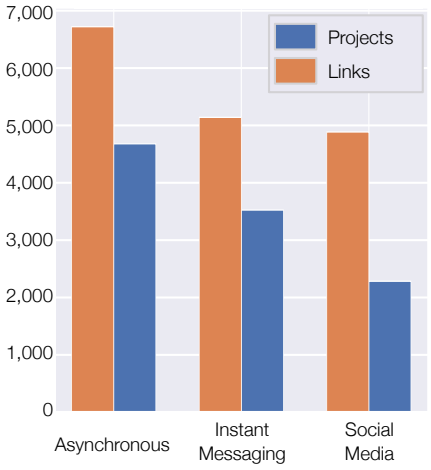


Figure 4.5: Platform types.

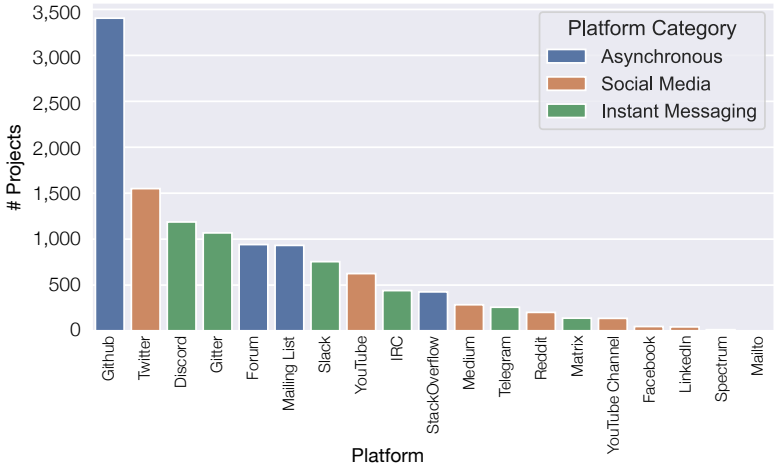


Figure 4.6: Number of projects linking at least one platform.

There are Instant Messaging platforms (*e.g.,* IRC, Slack, Discord), where communication can happen in real-time. In Asynchronous platforms (*e.g.,* forum, mailing list, GitHub issues or discussions), communication usually takes some time to be processed and made available to other community members. The boundary between the two types has been blurred in the recent years by the technological improvements to the supporting infrastructure.

We also considered Social Media platforms (*e.g.,* Facebook, Twitter, Youtube). While their technical features can overlap with the other types, their huge user-base and ease of forming social connections make them stand out. Table 4.3 shows a complete list of the considered platforms with their description.

A project readme can have multiple links to a single platform. This is particularly true for Social Media, where Twitter accounts of the main contributors or maintainers are all referenced.

In Figure 4.5, we see that the number of projects that use a specific platform is significantly lower than the number of links. For example, project *OpenAPITools/openapi-generator* [331] mentions 20 different Twitter accounts and 17 YouTube resources.

The identified categories are only a rough means to group similar platforms. In Figure 4.6 we show the number of projects having at least one reference to a specific platform.

Table 4.3: Communication platforms.

Platform	Description
Discord [319]	Voice, video, text messaging multimedia platform.
Facebook [322]	Social media and social networking service.
Forum	General category for web based discussion sites.
GitHub [323]	GitHub infrastructure for project development.
Gitter [321]	Voice, video, text messaging multimedia platform.
IRC	Text-based instant messaging chat system.
LinkedIn [324]	Business social media & professional networking.
Mailing List	E-mail based communication among recipients.
Matrix [325]	Communication protocol implemented by clients.
Medium [320]	Online publishing platform and social journalism.
Reddit [326]	Social news aggregation, rating, discussion, and multimedia sharing.
Slack [318]	Voice, video, text messaging multimedia platform.
Spectrum [308]	Text-based web instant messaging chat system.
StackOverflow [327]	Question and Answer website.
Telegram [328]	Voice, video, text messaging multimedia platform.
Twitter [329]	Social media and social networking service.
Youtube [330]	Video hosting and sharing platform.

Given our initial input set, it is not surprising to find GitHub to be the most referenced: The infrastructure is integrated enough to warrant support for the community with its own Issues and Discussions systems. This uniform consensus is followed by a more fragmented mix of Twitter, Discord, Gitter, Forums, Mailing Lists, and others in decreasing order of "popularity." Far from being irrelevant, these platforms are used by hundreds of projects exclusively or in synergy. The next section sheds light on these synergies, complementarities, and on the competition between similar platforms in the documentation landscape.

4.6 Coexistence and Competition

Communication platforms can have different features and cater to different audiences. To cover development or users' needs, projects can leverage multiple media at the same time. What choices are made by developers in terms of number and variety of platforms included in a readme?

Figure 4.7 depicts a non-exhaustive list of examples of overlaps between communication platforms (extracted with the REs presented in Section 4.3.4) used exclusively and side-by-side.

Around 38% of projects that use Discord or Slack also include GitHub Issues (Figure 4.7a).

GitHub Issues can also be used without an explicit link in the readme, as just a tab of the project, if enabled. Some platforms may be implicitly assumed to be available even if not present in the readme.

Overall, 2,105 out of 3,208 projects (66%) using GitHub Issues, also have other communication platforms referenced in the README. Similar ratios are found, for example, for Discord with 801 out of 1,187 projects (67%).

Multiple communication platforms of different types can and do coexist.

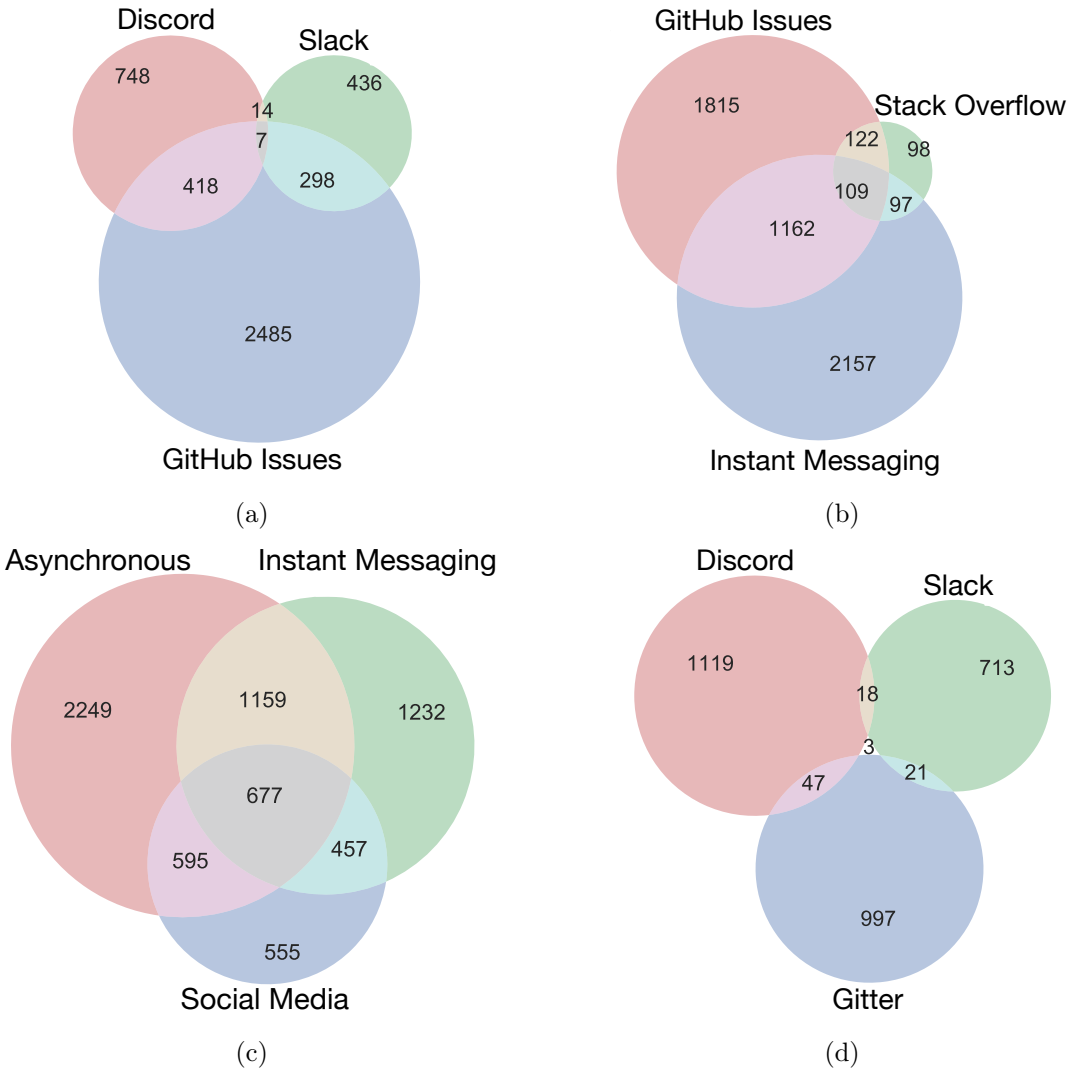


Figure 4.7: Communication platforms overlaps.

GitHub has significant overlaps with the whole category of instant messaging, and with specific asynchronous platforms (*e.g.,* StackOverflow, see Figure 4.7b). However, 1,270 projects rely only on the integrated support provided by GitHub.

In general, if we consider the three main categories, we find that asynchronous platforms are used exclusively in 48% of projects, instant messaging follows with 35%, and social platforms seem the most frequently used as a complementary option (76%, see Figure 4.7c).

It is not clear if different categories are mutually exclusive and why in a considerable amount of projects they tend to be used in conjunction.

This analysis should be complemented by how the user-base is distributed over these platforms.

4.7 Instant Messaging: A Deep Dive

What makes instant messaging platforms appealing to developers? The steady growth in the number of projects including at least one platform of this kind is a piece of evidence supporting the need for fast and rich communication.

Instant messaging platforms fulfill a very specific role: Providing communication in real-time, possibly with rich-media sharing capabilities (*e.g.,* links, videos, files), and Voice over IP conferencing (*i.e.,* VoIP). Two instances of these platforms are seldom found together. Similar characteristics, audiences, and usages make competition the prevailing paradigm.

Figure 4.7d shows that 97% of projects opting for these platforms choose between one of the three alternatives. Three projects include links (see Section 4.3.4) to all the platforms and also other instant messaging (*e.g.,* Spectrum), but only *PowerShell/PowerShell* has a significant Discord community (more than 10k members).

Gitter, Discord, and Slack are selected by projects as alternatives, very seldom co-existing. This can be a possible strategy for successful projects not to spread their community too thin over multiple platforms with similar capabilities.

4.7.1 Gitter, Discord, and Slack: A Timeline

Based on readme history and mining links for each version of the readme (Section 4.3.5), for each project, we look for the first appearance date of Gitter, Discord, and Slack (Figure 4.8).

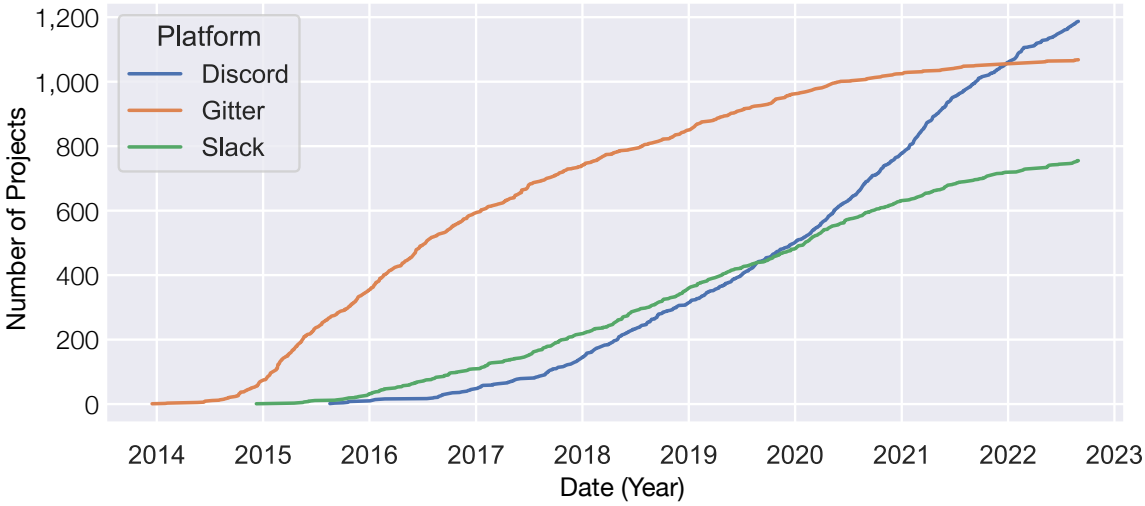


Figure 4.8: Timeline of cumulative adoption date of Slack, Discord, and Gitter.

Gitter appeared for the first time at the end of 2013, followed by Slack in 2014, and then Discord 8 months later. All the platforms show a "ramp-up" period of slightly more than one year after their appearance, followed by a steady growth at different rates. Gitter (in mid-2020), and Slack (in 2022) reached a *plateau* where, in the last year, just a handful of projects added them to their communication platforms. Discord, on the other hand, is still growing significantly.

Since the beginning of 2020, Discord consistently outperformed Slack in terms of number of new projects adopting the platform for their community (Figure 4.9). The monthly growth rate has been higher than the highest for Slack in the previous years. It has also been at higher levels more consistently and for a longer period.

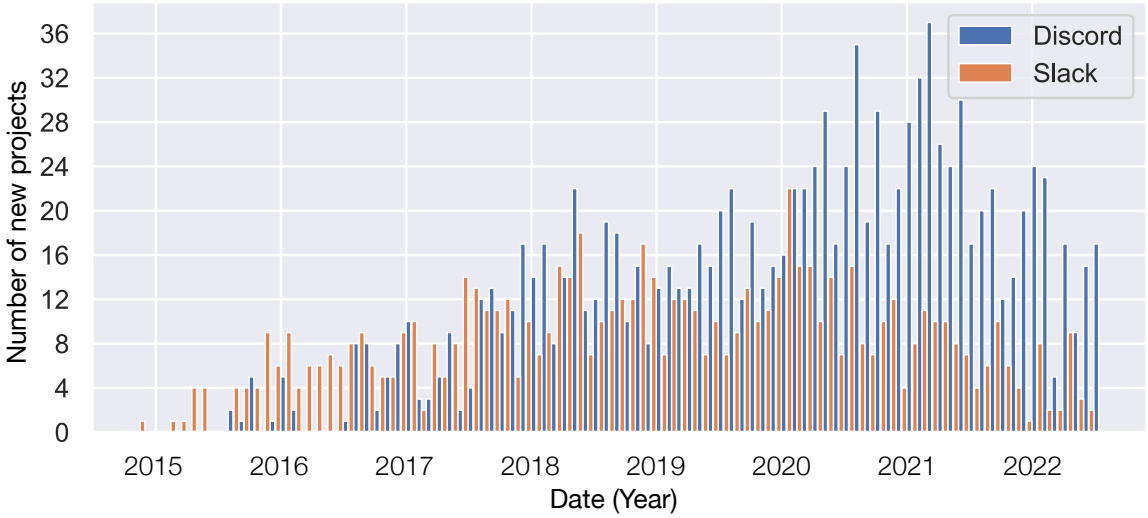


Figure 4.9: Monthly new projects adopting Discord and Slack.

The comparison between additions of Gitter and Discord (Figure 4.10) shows a similar or even more evident tendency. The decline of the former and the growth of the latter are almost perfectly mirroring each other.

> *While one platform stops being added to projects, another is on the rise. This happened in the past and is bound to happen again in the future.*

There is no guarantee that the example of the Spectrum platform we highlighted at the beginning of this chapter will be followed when Gitter goes out of fashion. It is also possible that the entire history of discussions, bug fixing sessions, and design decisions will just disappear.

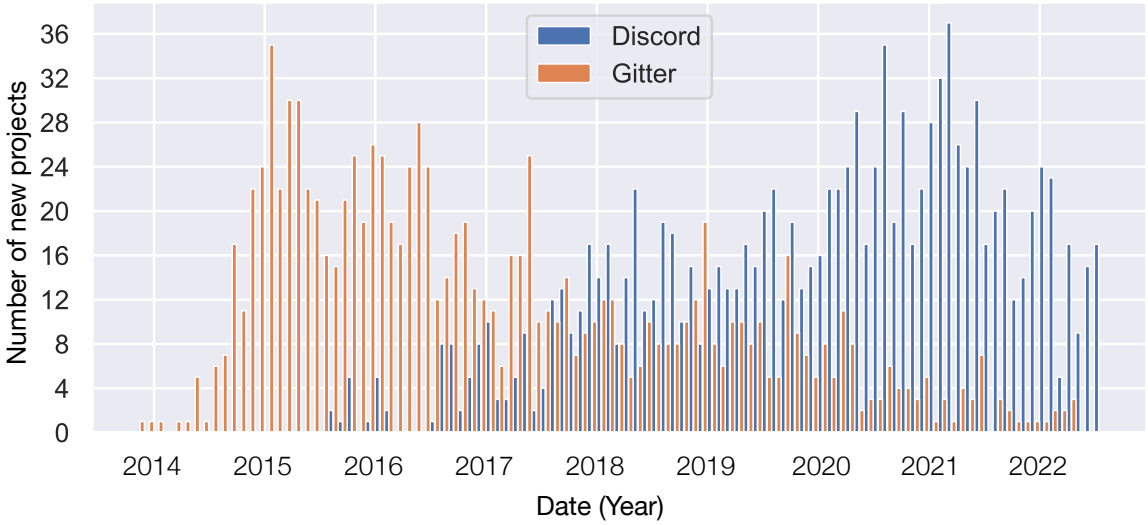


Figure 4.10: Monthly new projects adopting Gitter and Discord.

4.7.2 Throughput and Volatility

We investigated four Discord communities. Reactiflux, Vue Land, and Angular.js are respectively tied to React, Vue, and Angular (web development frameworks). We compared them with each other and with the Discord.js community (Discord bot development).

In Figure 4.11, we take a sample period of three months (*i.e.,* May–July, 2022) and we show the high variability of the average number of messages per day, in absolute value and normalized by the number of members in a server. This might be due to a number of factors, but we are interested in the sheer scale of the messages exchanged daily on those platforms.

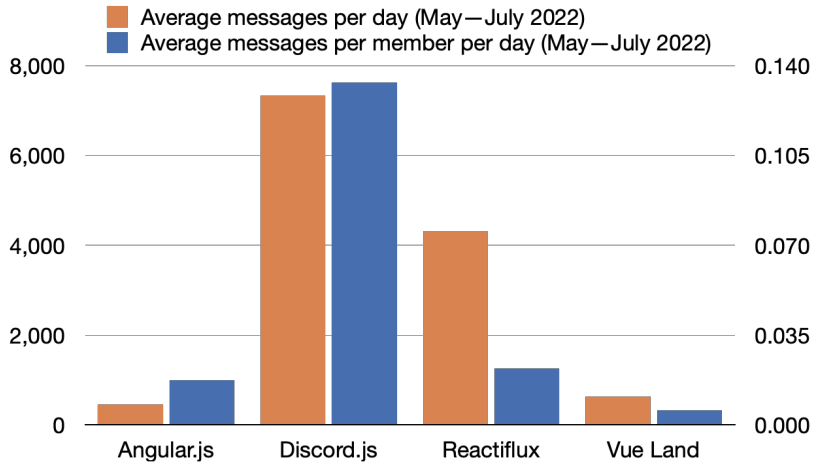


Figure 4.11: Messages per day and average messages per day per member from May to July 2022 for four example Discord servers.

Around 550 messages are exchanged per day in Vue Land and Angular.js. In Discord.js, instead, users exchange 305 messages *each hour*, totaling more than 7k messages a day.

The throughput of these servers means information is lost if one does not pay attention to notifications. Only a few messages are visible at a time and they scroll up quickly, putting full conversations behind the event horizon in a matter of minutes. Alert filters and community policies (*e.g.,* forbidden mentioning of server wide tags) can only partially mitigate this problem.

There is a trade-off between losing potentially interesting discussions and being constantly interrupted by notifications. This is the choice many modern developers face when dealing with these kinds of communities.

Application of summarization, visualization, and information retrieval techniques is fundamental to deal with scalability problems of these platforms.

4.7.3 Community Sizes

Discord communities in our dataset vary in size between 2 and 500,000 members. Figure 4.12 depicts Discord community size with respect to project age (*i.e.,* days since creation).

This should be investigated more in-depth to see if it is a breakpoint at which particular actions should be taken to keep the community growing.

Extraction of Slack community sizes has proven more difficult due to the high variance in invite page formats. Gitter does not even have an invite page to scrape, and, to the best of our knowledge, the community size cannot be automatically retrieved.

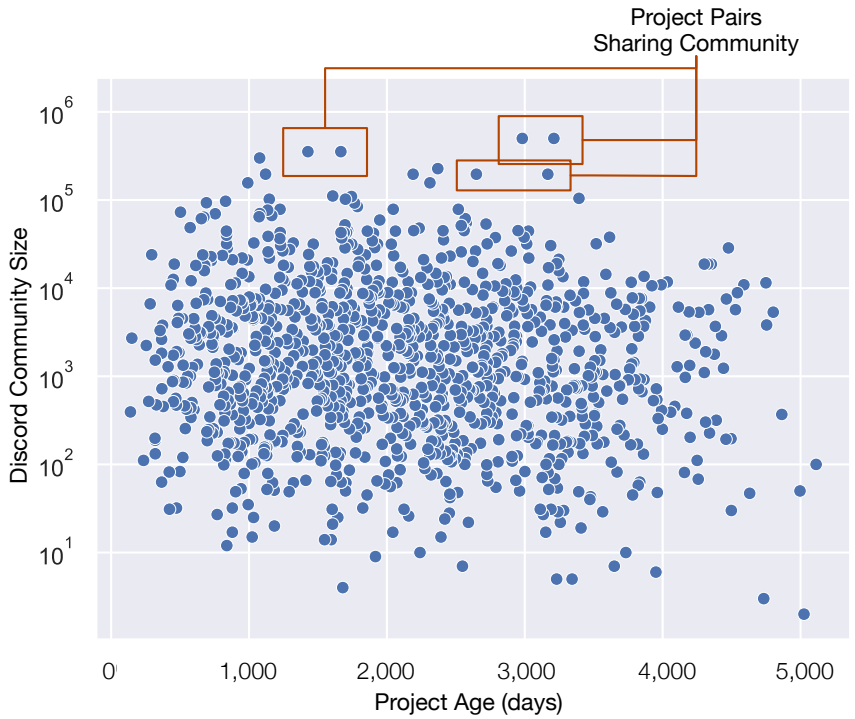


Figure 4.12: Discord community size with respect to project age (days from creation).

4.7.4 Different Projects, Same Community

Being in the same Discord community means sharing the same server (i.e., each project has a link in the README, possibly with different formats, but pointing to the same Discord server). We consider this a case of "different projects, same community".

Figure 4.12 shows horizontal pairs in the top part of the scatterplot, suggesting that different projects might share the same community: Our initial hypothesis that "same size of the community means same community" might not apply, especially for smaller communities. Nevertheless, it is unlikely for two different large communities to have the same number of members at the same time. We manually inspected the projects in those pairs and they are indeed different projects referring to the same wider community. For Discord we could reliably use the community name to confirm our hypothesis, as parsed from the invitation metadata (Section 4.3.4).

Figure 4.13 shows how many projects share a community with respect to community size.

*Further analysis can show if projects are tightly coupled (*e.g., *different projects from the same organization, new major versions of the same project) or if different projects have an underlying reason to cater to the same audience.*

4.7.5 Technical, Social, and Ethical Challenges

Some community platforms are public, some allow anonymous access, some require a form of registration or access permission. We found communities with automatic procedures to accept new members but, in general, it is hard to devise an automatic "agent" to explore all of them.

The sheer amount of customization that is possible, even in a simple invite landing page of Slack, has been an obstacle to getting reliable data about communities lying behind those pages. Exploring a larger and more varied sample could shed light on platform dependent similarities and differences to build a robust approach.

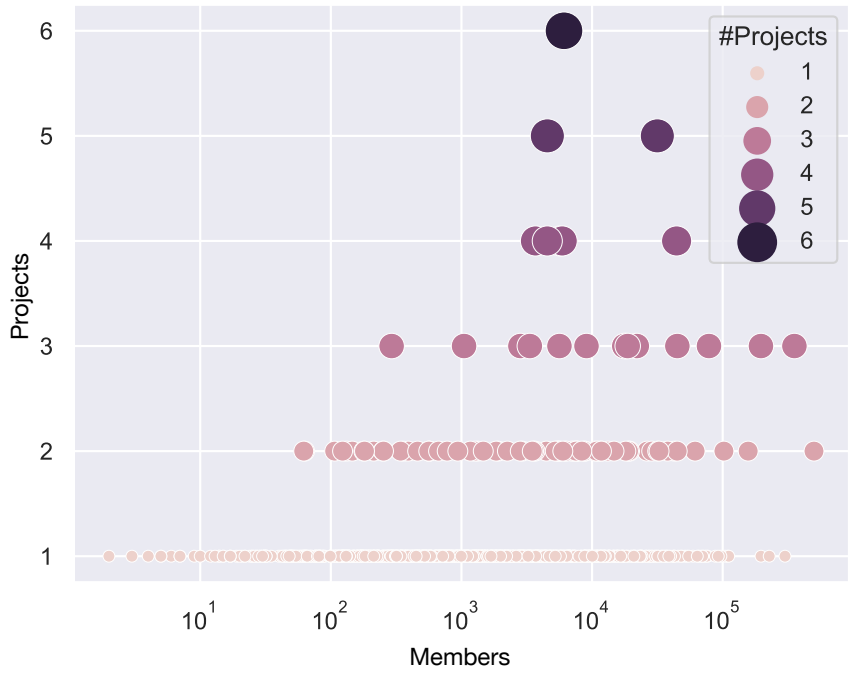


Figure 4.13: Projects referencing the same community.

Machine readable APIs for communities can greatly benefit research in this field and open new possibilities, e.g., *the automatic migration of community-generated content to preserve the history of projects when the underlying technologies evolve.*

Socio-ethical challenges

Usually, communities are digital aggregations of people's thoughts, ideas, rants, strengths, and weaknesses. Collecting such information can be seen as poking inside someone's house. One can do it if having legitimate reasons to do so. One can be welcome if providing benefits for the community. But one can also be faced with concerns about privacy and legitimate use of collected information. This is what happened when we tried to dig deeper in some Discord servers.

Companies owning the platform sometimes are more keen to share their data than administrators of communities they host. While Slack has a monetization policy tied to its history retrievability, Discord allows unlimited access to a wealth of historical information via its API.

A big role in the extensibility of our study is played by the attitude of administrators of interesting communities. While information might be public (*i.e.,* anyone with a Discord account can automatically join a server and browse its entire content), to comply with Discord's Terms of Service we need to ask for permission to add a bot to extract useful information for DwarvenMail. In this crucial step a fundamental role is played by personal beliefs and perceptions of the benefits of such a bot by the administrators and the community itself.

Accessibility of information is ever more beyond the fence of what is technically possible, towards the barrier of what is ethically and socially accepted.

4.8 Validity and Limitations

Our analyses are based on a dataset of public GitHub projects as the only source. This poses a threat to the generalizability of our results with respect to the type of projects hosted on GitHub. Open source projects developed in this social coding style might differ significantly from closed source projects developed by a small team of hired developers. There are also no guarantees that the results presented can be generalized to projects hosted on other similar repositories.

The current study presents a limited generalizability with respect to the format of readme files. Although our sampling procedure (Section 4.3.3) ensures generalizability with respect to the project's main programming language, different readme file formats could provide different link types and formats not fully captured by our analysis.

We found evidence of more than 15 different readme formats. While most share a similar structure for external links, systematic analysis of these formats could improve the generalizability of the results.

Limiting the extraction of the documentation landscape to what is reachable from the main readme file (*i.e.,* ignoring links to other readmes in submodules of a project) is a threat to construct validity. This threat is partially mitigated by the magnitude of the phenomenon we highlighted, emerging despite the limited scope, and calling for discussion and further investigation (*e.g.,* considering auxiliary documentation sources as starting point to map the landscape).

Our analysis benefits from verifying the validity of links whenever possible (*i.e.,* if the resource referred by the link is still available we expect an `HTTP 200 OK` response). When mining GitHub we verified the links we found in a two step process. The time interval between the first pass for scraping and the second pass for verification was short enough to guarantee that most links were in their intended state. Obsolete links may be possible and are part of the present study.

*The analysis lacks accuracy when links are redirected or reused. Moreover, in the effort to reconstruct link patterns for previous standard link formats of some platforms (*e.g., *Slack), we adopt a conservative approach. If the format follows reasonable patterns it is accepted as a valid link in the history of a readme file. We have no guarantee nor a way to discover if the link was valid in the past.*

The only possibility to study the evolution and validity of such links is to constantly monitor readme files and their changes over a period of time. Link validity can be checked as soon as the change in the readme is triggered. This kind of study is outside of the scope of our work.

Semantic analysis of the sources could improve relatedness, reducing false positives. Automatic link validity and topic relatedness should be investigated.

We analyze links that are not visually represented in the rendered readme. This threat to our conclusions validity is partially mitigated by the low frequency of such occurrences. We found only 3 non-rendered links in 2 manually annotated projects (0.1% of links, 3.3% of projects).

We did not perform an analysis of the relationships between project types, their intended audience, and the resulting documentation landscape. This could provide insights on the landscape for projects of different natures and maturity stages.

4.9 Summary

Classical software documentation is being replaced by "communication". At least in open source software on GitHub, it is supported by a plethora of platforms characterized by high throughput, volatility, and heterogeneity. The original vision of on-demand developer documentation [193] advocated for a paradigm shift. A shift did happen, but it was not in the direction originally foreseen by Robillard *et al.* The new communication platforms bring new challenges and opportunities for modern software documentation. It is time to shed light on new forms of documentation. A comparison with classical documentation and where it survives, unscathed by the new media and the needs of modern software development, might help rethink the role of documentation itself. Research efforts in this direction can help maintain documentation useful for software comprehension, maintenance, and evolution, independently of the form it will take.

To achieve this we need a better understanding of the phenomena occurring to software documentation sources. Our work scratched the surface of what emerged as an heterogeneous, complex, and ever-changing documentation landscape, a *terra incognita* of possibilities and threats.

With our deep dive in instant messaging sources we showed insights, promises, but also threats and limitations. In particular, Discord communities, their sizes, and their ability to cross the boundaries of a single project prompt for further research in this direction. Investigating these aspects across multiple source instances could help to identify cross-source patterns.

Finally, we focused on the interplay between content type (what), source type (how), and source instance (where). These are the three main dimensions of the software documentation landscape. Typically, each source instance promotes a specific combination of form and content, targeted at a specific audience for a specific purpose. Certainly not all the combinations are possible. Understanding which source instances are actually used in practice and for which reasons could help devise guidelines for practitioners, especially for sources with similar characteristics in one or more of these dimensions.

5

Analyzing the Evolution of the Software Documentation Landscape

Studying the software documentation landscape in small chunks provided depth of analysis. Our deep dive into the instant messaging sector of the landscape showed an interpretation of the paradigm shift affecting modern software documentation.

The evolutionary analysis of the documentation landscape at larger scales shows how this holistic interpretation can encompass a whole category of artifacts, practices, and documents. UML diagrams represent one of the most relevant, long-lasting, superficially consistent, and underutilized documentation source. The perfect candidate for our analyses.

We analyze the formats, technologies, and practices abandoned since years, the ones currently used by a large part of the software engineering community, and the emerging ones. We compare them highlighting trends supported by empirical data. These elements are at the core of our study of the evolution of the UML documentation landscape, which elicits numerous insights.

The disconnect between academia and industry regarding UML teaching and usage, respectively, is just the tip of an iceberg currently hitting the UML landscape. To avoid playing the role of the Titanic, UML needs to address the shortcomings of its new and popular human-readable textual formats (*e.g.,* PlantUML, Mermaid), rethinking its mechanisms (*e.g.,* layouting) without giving away what makes the new formats so appealing and usable.

5.1 Investigating the Past, Present, and Future of UML Documentation in Open Source Software

Andrew Watson said that the history of visual modeling can be divided cleanly into two eras, "Before UML" and "After UML" [259]. The period before UML was a time of division and strife, named the "Method Wars" era [7]. Booch, Rumbaugh, and Jacobson, a.k.a. the "Three Amigos", creators of three popular object-oriented development approaches [34, 109, 207], combined their efforts towards a single, Unified Modeling Language (UML) [81]. The adoption of UML as a standard by the Object Management Group in 1997 marked the end of the method wars and the beginning of the "After UML" era.

UML is one of the most prominent modeling languages for software design and development [169]. It has been used for automatic code generation for embedded systems [159], to create workflow diagrams for business processes [106, 122], to design [169], measure [56], maintain [77], comprehend [214], and reuse [14, 15] software systems.

We focus on UML usage in *designing and documenting* software systems.

Despite the apparent popularity of UML in academic publications, its history hints at a progressive stagnation. Minor revisions of the UML specification have been issued with an average yearly pace between 1997 and 2003 [332]. UML 2.0 took over two years to be released, losing the momentum. Last but not least, UML 2.5.1, the latest version of the specification, dates back to December 2017 without any updates in more than six years.

One of the main shortcomings of UML from a practitioner's perspective is the availability and quality of its tool support [168]. UML's decline in popularity since 2004[1] exacerbated the problem. This decline culminated with the discontinuation of language support in one of the major Integrated Development Environments (IDEs): Microsoft Visual Studio removed UML support in 2016 due to underutilization by clients, to focus development efforts on features deemed more relevant for a larger share of the user base.[2]

Given the benefits of UML diagrams for system comprehension, design, and documentation (see, for example, UML studies with the Lindholmen dataset [194]), we aim to understand why UML is still underutilized in practice, focusing on OSS. We investigated ∼13k OSS repositories to find those containing UML, identifying commonalities and differences. After tagging all UML files through their extensions and contents, we found that only 4.2% of the repositories in our dataset (552 out of 13,152) contained UML diagram files in the last 20 years. In the same time-span, UML file formats have significantly changed. Our findings show the progressive abandonment of dedicated UML graphical tools first, and changes in the role of IDE plugins later.

More importantly, we found empirical evidence of a resurgence of UML, after 2017, coinciding with the advent of human-readable, versionable, and easily maintainable text-based UML files (*e.g.,* PlantUML). This resurgence did not modify the natural tendency to neglect documentation and its maintenance. We conclude by analyzing the relationships among UML, repository activity, community, and human factors, focusing on UML artifact authors and their commits.

UML's decline in popularity and language stagnation are now countered by a renewed interest in human-readable text-based UML. Dropping special tools in favor of generic text editors, these formats introduce new challenges. For example, each tool uses a slightly different subset of UML, sometimes with an ambiguous specification. Moreover, an effective and consistent layout of visual elements is usually impossible with these tools, severely impacting their effectiveness [206].

We argue that Watson's split needs to be revised and that a third era has already started. Only a convergence between language evolution, textual human-readable UML standardization, and improved tools can address these issues for a (new or real?) golden age of UML.

5.2 Research Questions

Given the advantages that the use of UML should provide, according to the literature, and the fact that a great effort is spent in teaching the language in universities, the fundamental meta-question of our analysis is:

META-QUESTION

Why is UML still underutilized?

As any why question has no imminent answer, we focus in the following on four distinct research questions pertaining to the format, diffusion, demographic, and the designers of UML diagrams, which we answer through an empirical study.

[1] See `https://trends.google.com/trends/explore?date=all&q=UML&hl=en`
[2] See `https://www.infoworld.com/article/3131600/`

We define a strategy to semi-automatically capture UML artifacts, based on file formats. This necessary step to collect data on the use of UML leads to our first research question:

RQ$_1$ — FORMAT

What formats are UML diagrams found in?

With a dataset of 20 years of history of semi-automatically gathered UML artifacts, we focus on the evolution of UML usage trends at large, from the beginning to the current state:

RQ$_2$ — DIFFUSION

How widespread is UML in open source software?

Different characteristics of the projects using UML, with respect to activity (*e.g.,* commits, releases), participation (*e.g.,* pull requests, contributors), or community engagement (*e.g.,* watchers, stars), could be used to advocate for the ubiquitous adoption of UML. If UML would contribute positively to certain metrics (*e.g.,* participation), its use should be further encouraged based on these benefits. Otherwise, the meta-question might be dismissed by just saying that UML is under-utilized because its practical usefulness is overestimated. Which leads to:

RQ$_3$ — DEMOGRAPHIC

What types of projects include UML diagrams?

Finally, our focus shifts from repositories to contributors to analyze the characteristics of key figures in the lifecycle of UML artifacts. If no significant difference can be found in the repositories, can human factors related to UML contributors (those actively working with UML artifacts) motivate the under-utilization of UML? Hence our final research question:

RQ$_4$ — DESIGNERS

Who is creating and maintaining UML diagrams?

5.3 Dataset Creation

We used the SEART GitHub Search[3] (GHS) tool from Dabic *et al.* [52] to gather an initial set of GitHub projects. After excluding forks, to avoid "polluting" the dataset with social forks [121, 266], we selected projects with at least 2,000 commits to eliminate toy repositories. We selected projects with at least 10 contributors to ensure the need for collaboration, which increases the utility of diagrams. We selected projects with at least 100 stars to ensure projects are considered useful by the OSS community.

Our dataset consists of 13,152 repositories, which we cloned locally on April 1, 2023. We performed a deep cloning for each project to analyze their entire history. After extracting with *cloc* [333] the number of lines of code (LOCs), we removed the projects with less than 10k LOCs.

cloc counts LOC in many different languages, we found 278 of them, for example C, C++, but also C/C++ Headers and JSON. We used *cloc* mainly to remove from our dataset emptied projects which kept the statistics we used as GHS filtering criteria.

[3]See `https://seart-ghs.si.usi.ch/`

We stored summary information about each repository in a PostgreSQL database. In Table 5.1, we provide descriptive statistics about the complete dataset.

Table 5.1: Statistics of the dataset with 13,152 projects.

Metric	Total	Min	Median	Mean	Max
Commits	125.0M	2,000	4,214	9,506	841 k
Contributors	*1.5M	10	63	112	13 k
Files	30.6M	7	849	2,330	128 k
LOCs	6.4 B	10,014	127,785	485,026	46M
Stars	42.1M	100	708	3,201	322 k

*Non-unique total contributors.

5.4 Mining and Analyzing UML Repositories

In Figure 5.1, we show an overview of our approach to mining and analyzing the repositories on which we base our study of UML evolution.

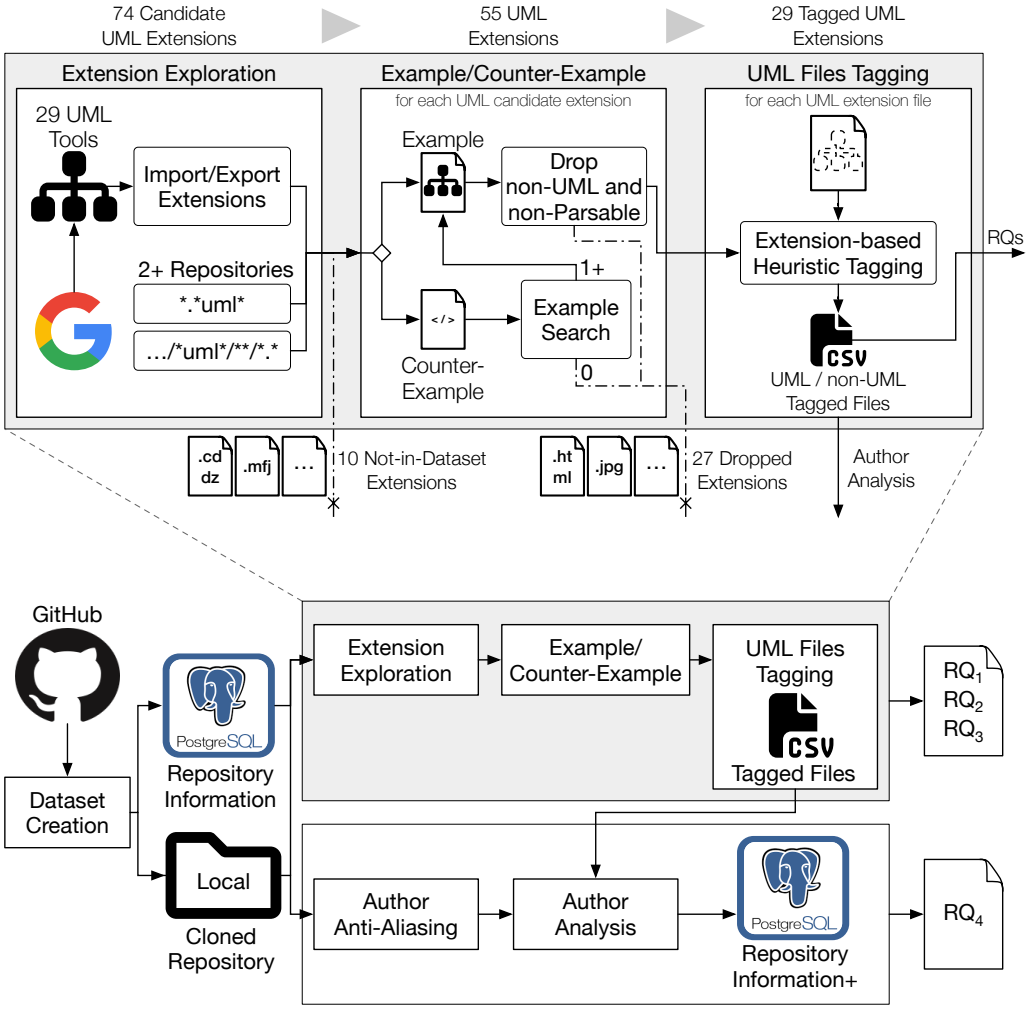


Figure 5.1: Approach overview (bottom) and detailed UML extraction (top).

We split the meta-question into four research questions (RQ_{1-4}) on the usage and the nature of UML in OSS repositories. For each research question we give an overview, then we present our methodology, results, and findings.

5.4.1 RQ_1: What Formats are UML Diagrams Found in?

File extensions are an efficient way to discriminate file types, but they are not sufficient to uniquely identify UML diagrams. Thus, to obtain a "handle" on UML artifacts useful to address all the RQs, we implemented strategies to manually find examples and counter-examples of UML diagrams for different file extensions. For each potential UML file extension (*i.e.,* those with at least a UML example), we devised heuristics to tag semi-automatically the files that contain UML. In the following we describe each step of the UML extraction procedure. In Figure 5.1 (top), we show a "zoomed-in" overview.

Extension Exploration. We created a list of candidate UML extensions using the following approach: We searched for "top UML diagramming tools" on Google and identified 29 popular tools for creating UML diagrams. We downloaded and installed each tool to determine the supported file extensions. We checked their import, export, and save functionalities.

We could not install or run 5 programs, for example, due to licensing issues. For those, we relied on the documentation or YouTube videos to determine which file extensions were supported. We complemented this approach by searching for all extensions with "`uml`" in the name or in any part of the file path and present in at least two repositories. After generating the list of candidates, we removed any extensions for which we found no examples in our dataset (*i.e.,* a tool's unused import/export format), along with any extension used by known non-UML file types (*e.g.,* .java, .jar, .am).

Example and Counter-Example. For each file extension we searched for an example and a counter-example. Extensions for which we could find at least an example are valid UML extensions. Those for which we could find at least one counter-example are not UML-specific. We randomly selected, retrieved, and manually inspected one file for each extension, to confirm whether it was a UML diagram or not.

We were left either with an example or a counter-example. If, for example, we had a counter-example for .txt files, we then searched all .txt files to find an example of UML. In this search we tried to reduce the number of files to be inspected before finding an example. If we were looking for a UML example, we first analyzed the files whose path contains the string "`/uml/`". Then we searched for the following keywords in the file names (case insensitive): *architecture, uml, diagram, sequence, class, usecase, state, activity, component, deployment, object, communication, composite, interaction, collaboration, package, profile,* and *timing* (part of this keyword list is comprised of the names of the 14 UML diagram types). We inspected these files first. For extensions with more than 150 files, we explored a statistically significant sample (confidence level = 95%, margin of error = 5%). Then, we removed extensions that are too generic (*e.g.,* .html, .md)[4] and formats whose content we cannot parse automatically (*e.g.,* .jpg, .png).

To recap, the procedure has multiple steps with branches (see Figure 5.1 right). If a randomly sampled file for an extension is an example (UML file with that extension), we mark the extension as UML (not yet exclusive). Then, we look for counter-examples for the extension to see if this is a UML-exclusive extension or if we can find both UML and non-UML files with this extension.

[4]The exclusion of these file extensions is a potential limitation of our study but it is mitigated by the fact that, to the best of our knowledge, no tools except EnterpriseArchitect can import HTML or MarkDown files to produce UML diagrams, rendering the collaborative maintenance of such artifacts impractical (see `uml-tools-extensions.md` in the replication package for a list of extensions importable and exportable by each tool).

If we find a counter-example, this is not a UML-exclusive extension. If the first randomly selected file is a counter-example (not a UML file), with no examples found in the second step, it corresponds to a format to discard (no UML files found with that extension).

UML Files Tagging. Some file extensions are not used for UML purposes exclusively (*e.g.,* .dia, .drawio, .graffle). We tag each UML file in our dataset not only based on its extension, but by semi-automatically parsing its content.

For each file with an extension coming from the previous step, we detect UML content via extension-specific regular expressions (after de-compressing files). For example, .ecore files are UML files for the Eclipse Modeling Framework if they contain either the word "EClass" or "EPackage". By devising such strategies we automatically tagged almost every UML file in the dataset. Where an automatic approach was not feasible, we checked the files manually. This was needed for 26 files of 3 extensions (.asta, .cmof, .yuml). This step consisted in a progressive refinement of the strategies until all files with all extensions resulted in a UML or non-UML tag. Table 5.2 shows the UML extensions considered in the rest of the study.

Table 5.2: Repositories with tagged UML files.
(Out of 552 repositories with UML, see Section 5.4.2)

Extension	# Repos	Extension	# Repos
.puml	160	.diagram	8
.uml	100	.pgml	8
.dia	67	.prj	8
.xmi	60	.zuml	7
.plantuml	48	.asta	6
.ecore	43	.session	6
.ucls	33	.iuml	5
.zargo	32	.mdzip	5
.uxf	30	.yuml	5
.mmd	26	.gliffy	3
.mdj	18	.platuml	2
.vpp	14	.umlprofile	2
.pu	10	.cmof	1
.umlclass	9	.ump	1
.argo	8		

There are two assumptions we made: (*i*) for each file we examine only its first version, assuming that the nature of a file does not change, only its content does; (*ii*) if multiple files with the same name and extension exist in different paths, we analyze only one of them, assuming that files with the same name have the same type. This simplification for the manual analysis impacts only identification of the tagging strategies. The applied heuristics tag each file in the different paths as UML or non-UML and each file is counted independently.

Results. Figure 5.2 shows the evolution of the most popular UML extensions by number of repositories using them, since 1999. For each year, we plot the number of repositories actively making changes (with at least one commit) to UML files for each extension. The red bars represent the top-5 extensions found in the most repositories for that year.

The .xmi, .argo, .pgml, .zargo, and .dia are the only UML extensions used from 2000 to 2003. The .zargo, .argo, and .pgml extensions are used by ArgoUML, and .xmi is a standard for many graphical UML tools. Dia is a diagramming tool.

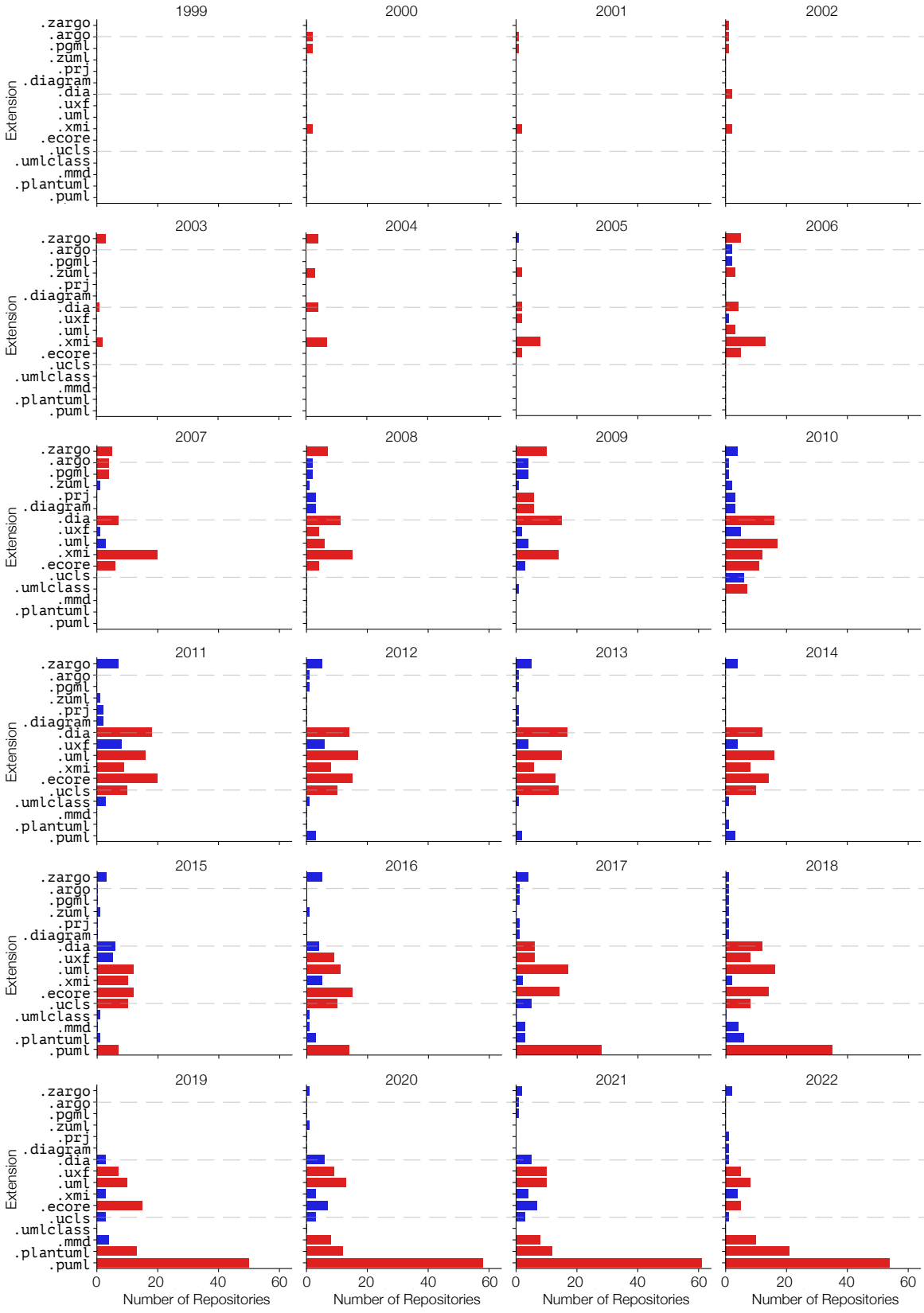


Figure 5.2: Number of repositories for most popular UML files modified each year. The top-5 most popular extensions for each year are highlighted in red.

It is worth noting again, in this context, that extensions such as .xmi can also be used for non-UML content. As described previously in this section, we consider only the UML-tagged files for each extension. The regular expressions we devised count only .xmi files with actual UML content. The same applies to other generic extensions (*e.g.,* .dia, .ecore).

2004 to 2007 is still dominated by .xmi and .zargo, but .dia's use increases and .ecore appears. The Eclipse Modeling Framework (EMF), an Eclipse plugin for graphical modeling and model-based automatic code generation, uses .ecore files.

From 2008 to 2015, .xmi, .dia, and .ecore remain popular while .ucls, .uxf, and .uml start to become popular. The .uml extension contains many types of UML files, including Ecore/EMF, XMI, and PlantUML files. The .ucls extension is used by the ObjectAid UML Explorer Eclipse plugin. The .uxf extension is the UML eXchange Format (similar to XMI).

From 2016 to 2022, the previously popular extensions are supplanted by .mmd (Mermaid), .plantuml, and .puml (PlantUML). In 2022, the two text-based UML diagramming tools (*i.e.,* PlantUML and Mermaid) are the most used.

Given the above, we can see three main periods. Early on, from 2000 to 2007, we see the usage of graphical UML modeling tools and generic diagramming tools. From 2008 to 2016, Eclipse plugins took over as the most popular. Finally, from 2016 to 2022, we see the rise of text-based UML. Interestingly, PlantUML's use did not rise until 2016, despite being released in 2009. Similarly, Mermaid released in 2014, gained popularity in 2016, and made it into the top-5 extensions in 2020. One event that possibly contributed to this shift is the release of Visual Studio Code (VSCode) in 2015.

Given how popular Eclipse plugins were from 2008 to 2016, developers liked UML modeling tools to be integrated into their IDE. Since its release in 2016, VSCode has become the most popular IDE for developers.[5] It has extensions in its marketplace (*i.e.,* plug-ins) that support rendering and editing UML diagrams in PlantUML, Mermaid, and UMLet formats.

From 2016 to 2022, we see a huge uptick in the overall number of repositories actively working with UML, mainly due to the rise of PlantUML. This could be due to the advantages that text-based UML modeling tools offer. Compared to graphical UML modeling tools, it is arguably easier to version control text-based UML diagrams, for example, having meaningful *diffs* that can be checked for correctness during code reviews.

Summarizing: RQ_1 investigates how UML tools and formats evolved in the last two decades. We present the trends, highlighting the novel ones and the recent history. The *underutilization* of UML emerges from the number of repositories containing UML artifacts. The *Why* of the meta-question is related to the tools and formats themselves. For example, a fragmented ecosystem of tools has maintainability and interoperability problems, leading to underutilization (Section 5.5).

RQ_1 — Findings

- F_1 PlantUML has been the most popular UML diagramming tool since 2016, UML's resurgence year.
- F_2 Mermaid (*i.e.,* .mmd) surpassed .uxf and .uml extensions in 2022. The other human-readable text format becomes the second most popular after PlantUML.
- F_3 Text-based UML diagramming has supplanted UML graphical editors over the past few years.

[5] See `https://survey.stackoverflow.co/2022`

5.4.2 RQ_2: How Widespread is UML in Open Source Software?

For this RQ, we analyze the history of UML's diffusion to (re-)assess the current status of the practice, providing insights on the evolution that brought UML to be an under-utilized language. We look at the percentage of repositories that contain at least one UML diagram since their creation. We analyze the popularity and activity on UML files over the last two decades.

Method. After identifying candidate UML extensions and tagging UML files for RQ_1, we further restrict the dataset we first introduced in Section 5.3 to repositories containing UML diagrams, resulting in 552 repositories. Table 5.3 summarizes descriptive statistics about the dataset of UML repositories.

Table 5.3: Statistics of the dataset with 552 UML repositories.

Metric	Total	Min	Median	Mean	Max
Commits	7.9M	2,001	6,866	14,333	157 k
Contributors	*56.3k	10	62	102	441
Files	2.3M	60	2,038	4,254	60 k
LOCs	462.7M	10,179	302,268	838,141	13M
Stars	1.3M	100	533	2,427	80 k

*Non-unique total contributors.

Results. The usage of UML in our dataset is not widespread. We found only 4.2% of repositories (552 out of 13,152) that contained at least one UML diagram at some point in time.

We investigated whether the usage of UML diagrams has increased or decreased over time. Since the number of active repositories changes over time, we normalize the number of repositories with UML diagrams over the number of active repositories per year (*i.e.*, having at least one commit on the main branch in that year).

Figure 5.3 shows the percentages of repositories with UML diagrams and of repositories actively modifying UML files (at least one UML file commit in the year) since the year 2000.

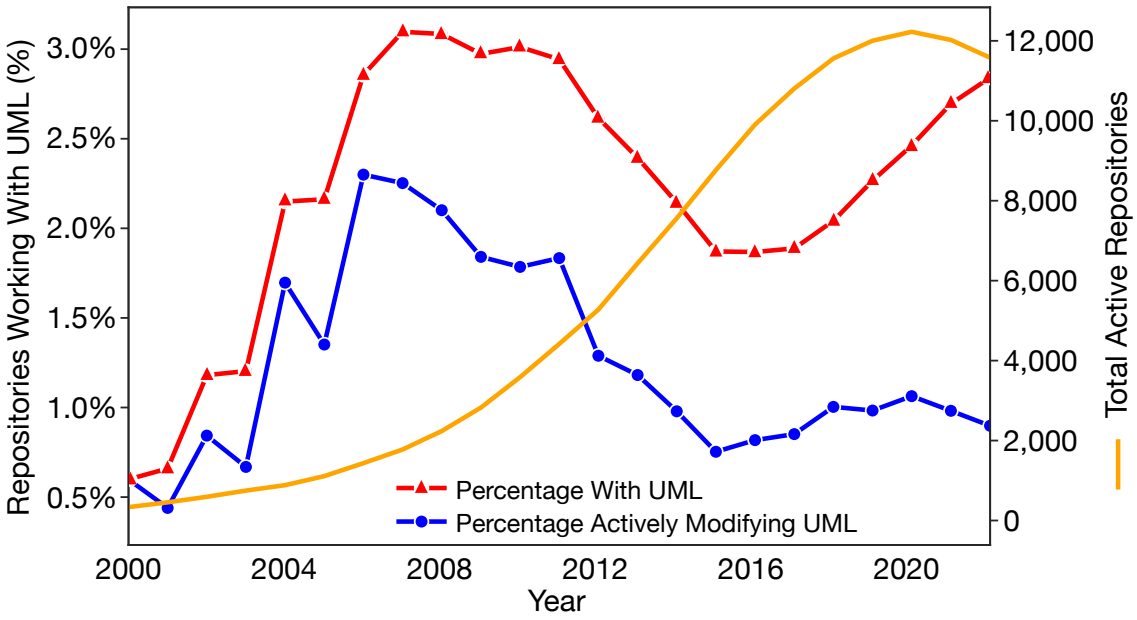


Figure 5.3: Percentage of repositories with UML.

At its peak, about 3.0% of active repositories contained a UML diagram. The fraction of repositories actively modifying UML is even lower (*i.e.,* at a maximum of 2.3% of repositories modifying at least one UML file in 2006). The current percentage of repositories actively using UML (blue, circles) hovers around 1.0%, slightly above the latest local minimum in 2015. Less than one third of the projects containing UML saw activity in UML files in 2022.

The overall percentage of repositories with UML (red, triangles) follows a similar trend before 2015, but it has been increasing since 2017. The use of UML peaks around 2007, but in the last five years the divergence between projects merely containing UML and those actively modifying it has been consistently increasing.

Summarizing: RQ_2 takes a step back and considers aggregate UML usage and maintenance data. We imply, as further discussed in Section 5.5, that if we do not take into account the shortcomings of the new wave of UML tools and formats, this resurgence wave might not bring UML out of the underutilization condition (*i.e., Why* UML will remain underutilized).

RQ_2 — Findings

- F_1 UML in OSS is not widespread, with a peak of only 3.0% of active repositories having a UML diagram.
- F_2 UML hit peak diffusion in 2007, then it decreased to below 2.0%, just to see a steady resurgence that is still going on since 2017.
- F_3 Usage and active maintenance of UML follow different trends since 2016. The first is increasing while the second stays almost stable in percentage.

5.4.3 RQ_3: What Types of Projects Include UML Diagrams?

Assuming that UML's presence should influence or be influenced by the development process, we hypothesize that a measurable effect should appear in some metrics related to the repository activity or to the community of developers contributing to the project. To confirm or refute this hypothesis, we analyzed what types of projects contain UML diagrams, comparing projects by their main language and 10 GitHub metrics related to activity and community.

Method. To analyze correlations between project metrics and the presence of UML artifacts, we compared UML and non-UML repositories with respect to the following metrics: *commits*, *contributors*, *forks*, *open issues*, *open pull requests*, *releases*, *stars*, *total issues*, *total pull requests*, and *watchers*. For each metric, we performed a D'Agostino-Pearson normality test to verify if the data was normally distributed. Since in both cases the data was not normally distributed, we used the Mann-Whitney U test on each metric to determine if there was any statistically significant difference. For those metrics where we found a statistically significant difference between the two samples, we used Cliff's delta to determine the magnitude of the difference.

We were also interested in possible correlations between the main language of the repository and the usage of UML. Therefore, we grouped UML repositories by main language.[6]

Results. UML diagrams can describe various aspects of a system. We hypothesized that repositories with higher activity and larger communities would be more likely to have UML diagrams. This was not the case. The groups of UML and non-UML repositories have a statistically significant difference on the number of commits, open and total pull requests, stars, and open issues (Mann-Whitney U test $p \leq 0.01$), but only the number of commits has a non-negligible effect (positive correlation with presence of artifacts, Cliff's delta $\delta = 0.30$, small effect).

[6] As reported by GitHub's language statistics.

We investigated the effect of the choice of programming language on the use of UML. In Figure 5.4, we show the percentage of repositories with UML by main programming language.

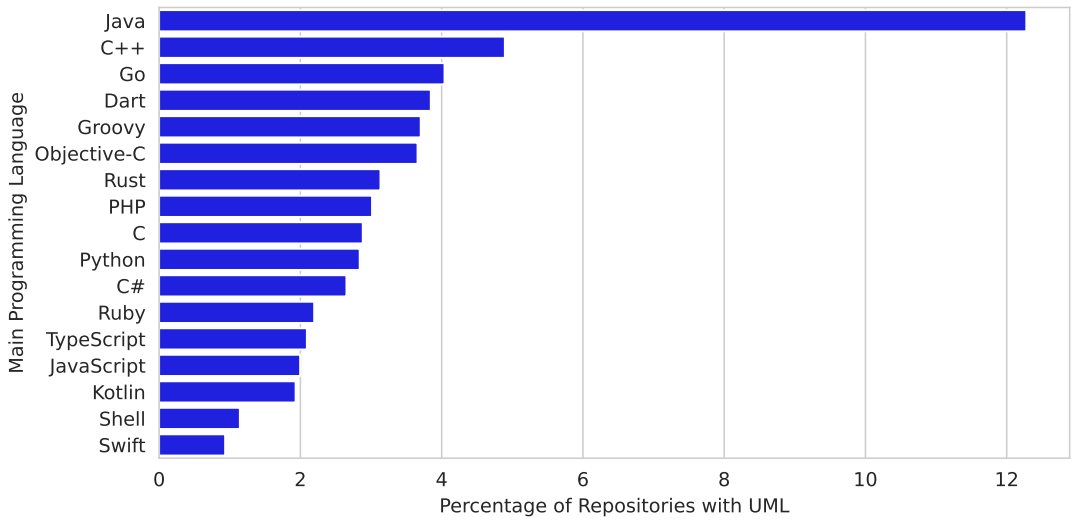


Figure 5.4: Percentage of repositories with UML by main programming language.

We see Java as absolute first by a large margin, followed by C++ and the others with less than half its popularity. The choice of programming language seems to affect how likely a repository is to have UML. Java repositories are three times more likely to have UML than repositories in other languages. The lack of a clear separation between languages, except for Java, makes it difficult to identify distinguishing characteristics among those more likely to have UML diagrams.

Java and C++, which support class-based object-oriented programming (OOP), are in the top spots. Following them, at number three, there is Go, which does not have classes. Then, again, we find languages which support class-based OOP near the bottom of the list, like Kotlin and Swift. Since UML was built to support OOP methodologies, it is also interesting to find C, which is not an OOP language, above Python, C#, and Kotlin.

Although we see a clear difference in the likelihood of a repository to have UML depending on its main language, we do not see any patterns in the types of languages for which this is more likely. As part of our future work, we plan to further investigate the relationships between all the project's languages (*i.e.,* not only the main one) and the use of UML.

Summarizing: RQ_3 investigates if the *Why* of the meta-question is connected to any significant improvement in relevant metrics. Apart from a slightly increased number of commits, the activity and community metrics we examined were not affected by the presence of UML artifacts in the repository, an argument in favor of the underutilization (*i.e.,* UML is underutilized because it does not improve developers' engagement with the project).

RQ_3 — Findings

- F_1 Repositories with UML have a slightly but significantly higher number of commits.
- F_2 Except for the number of commits, there is no difference in activity and community metrics between UML and non-UML repositories.
- F_3 Java is the most likely programming language to use UML, with 12% of repositories (4 times the average of the other languages).

5.4.4 RQ_4: Who is Creating and Maintaining UML Diagrams?

The use of UML diagrams does not seem to influence the metrics of activity in the repository we considered, but interesting relationships could be revealed considering the project contributors. This question needs a thorough analysis of proxies such as the contribution period (*i.e.,* as a proxy for involvement in the project).

We analyze creators and maintainers of UML diagrams compared to developers. We investigate how these roles overlap and the characteristics of authors with dedicated project management roles contributing to UML diagrams.

Method. To answer RQ_4, we needed to integrate authors' information with the list of tagged files, after removing author aliases [32, 88, 89, 127]. We expanded on the work of Gote and Zingg [89] by tuning the algorithm to GitHub repositories. We eliminated names and emails that we found were common placeholders in our dataset (*e.g.,* anon@github.com). We specifically ignored matches of the following names: *unknown*, *anonymous*, *anon*, and *none*. We also excluded emails containing the words *unknown*, *anonymous*, *devnull*, *noreply*, *none@none*, and *root@localhost*. In addition, we considered names in the domain of email addresses for those who host their own email accounts (*i.e.,* mail@jdoe.com). When extracting the email, we compare the domain instead of the base if the first or last name appears in the domain. The PostgreSQL database is augmented with the resulting derivative information (*e.g.,* author anti-aliasing, author analysis). A complete dump of the final database is available for replication and further exploration on *figshare* at `https://figshare.com/s/96cde375f69347469472`.

Contribution Period of UML vs. non-UML Committers. The first attribute we analyze is the average contribution period: The period between the first and the last commit of a contributor. In Figure 5.5, we show three box plots with the distribution of the average contribution period of UML and non-UML committers.

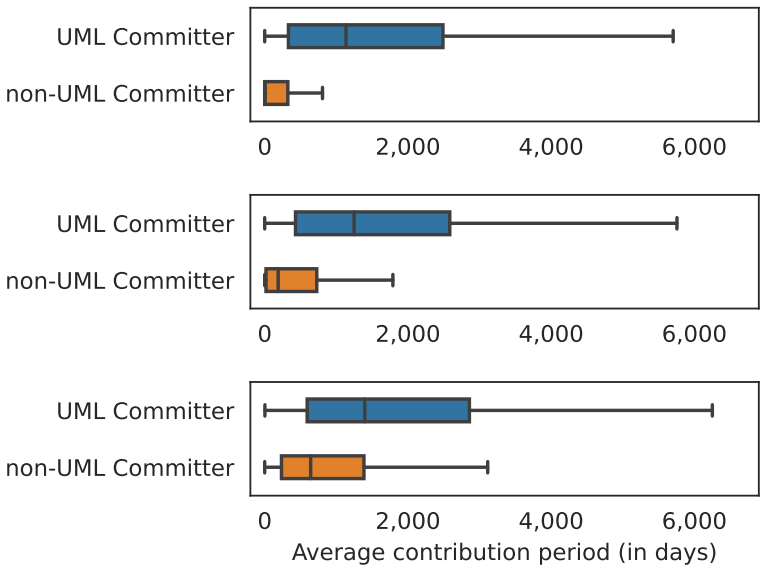


Figure 5.5: Average contribution periods of UML vs non-UML committers. All contributors (top), contributors with at least 2 commits (middle), and contributors with at least 10 commits (bottom).

The top box plot shows the distribution without filtering, and the middle and the bottom ones are obtained by removing contributors with less than 2 and 10 commits, respectively. The box plots of UML committers are only slightly affected by filtering.

On the contrary, the median contribution period of the non-UML committer box plots shifts to the right, moving from 0 days with no filtering to 801 days when filtering out contributors with less than 10 commits. This huge shift is due to the large number of occasional contributors in terms of source code. This phenomenon is almost non-existent for UML committers. For this reason, in the following analyses, we show only the results after removing contributors with less than 10 commits.

The median contribution period of UML committers is 1,735 days versus 801 days for non-UML committers. We confirmed that this difference is statistically significant (Mann-Whitney U test $p \leq 0.01$, Cliff's delta $\delta = 0.56$, large effect). UML committers contribute to the repository for a significantly longer period (more than twice).

In Figure 5.6, we see a scatter plot of the same average contribution. Each point represents a repository. Any point above the red line is a repository where the average contribution period of the UML committers is greater than the average contribution period of the other authors.

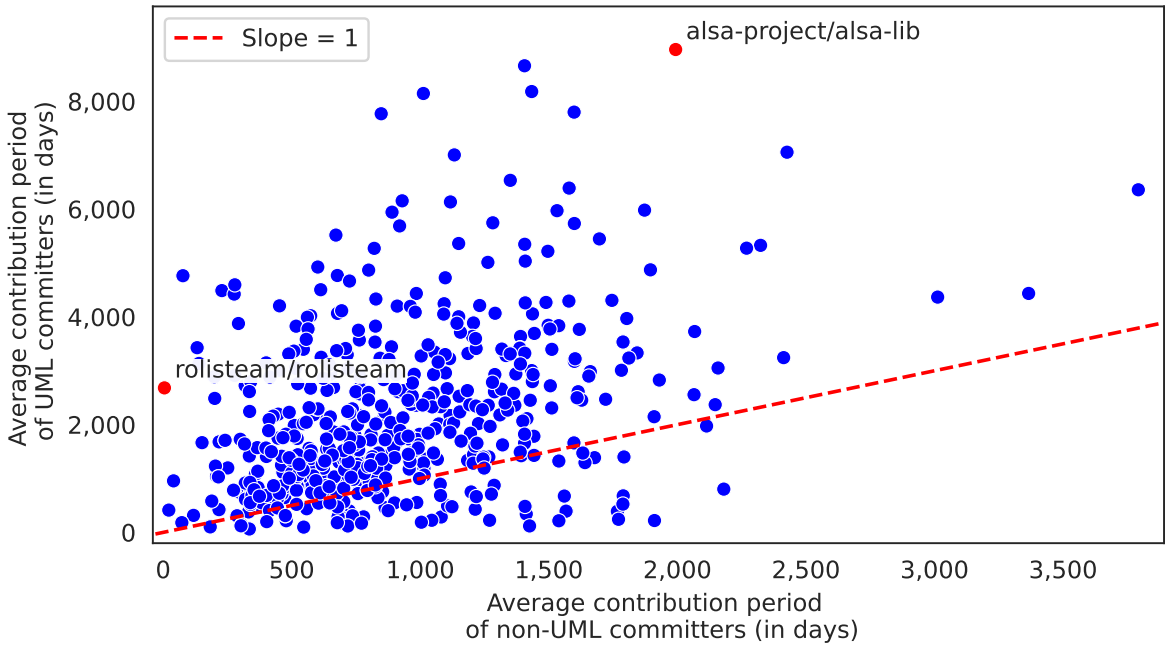


Figure 5.6: Average contribution periods of UML vs. non-UML committers.

We marked two outlier repositories in red in the scatter plot. The first is the one with the lowest average contribution period of non-UML committers while still having a high average contribution period of UML committers, *rolisteam/rolisteam*. The contributors for *rolisteam/rolisteam* can be seen in Table 5.4, with the UML committers marked with the icon.

We can see the repository has a single maintainer, *Renaud Guezennec*. Most of the remaining contributors had all of their commits on the same day (author contribution period of 0).

The main maintainer is the only person who has also created or modified any UML diagrams, which explains the high average contribution period of UML committers.

The other marked repository has the highest average contribution period of UML committers, *alsa-project/alsa-lib*. Table 5.5 shows the top-5 authors by number of commits.

This project is similar to *rolisteam/rolisteam*: Only one contributor modified any UML diagrams. However, in the *alsa-project/alsa-lib* repository, many other contributors have been active for a long time, making numerous non-UML commits.

Table 5.4: Contributors for rolisteam/rolisteam. (UML Committers)

GitHub Profile Name	Contribution Period	Commits
Renaud Guezennec	2,682	5,558
Vladar4	56	2
Etienne	8	3
Tyler Schmidt	8	3
Tomaz Canabrava	5	35
Milan Irigoyen	3	4
Paul Brown	0	4
Gissu	0	5
Ben Cooksley	0	4
Yann Escarbassiere	0	3
Patrick José Pereira	0	3
Grégoire Barbier	0	1
IBPX	0	1

Table 5.5: Top-5 contributors by number of commits for alsa-project/alsa-lib. (UML Committers)

GitHub Profile Name	Contribution Period	Commits
Jaroslav Kysela	8,962	1,967
Takashi Iwai	8,027	925
Clemens Ladisch	4,764	126
Takashi Sakamoto	2,870	151
Abramo Bagnara	1,227	355

Number of UML Committers vs. non-UML Committers. We investigate how many contributors in a repository are UML committers. Figure 5.7 shows a scatter plot of the number of UML versus non-UML committers.

In almost every single case, the number of non-UML committers is significantly higher (Mann-Whitney U test $p \leq 0.01$, Cliff's delta $\delta = -0.93$, large effect) than the number of UML committers (repositories under the red dashed line).

We investigated more in detail a few different cases: *iluwatar/java-design-patterns* with a ratio close to 1.0, *umple/umple* with more UML than non-UML committers, *kubernetes-sigs/cluster-api* with more non-UML committers (but still more than 20 UML committers), and *embox/embox*, as representative of the typical case.

The *iluwatar/java-design-patterns* project consists of an educational repository used to teach Java design patterns. Every pattern follows this template: A Java example, a UML diagram, and a readme with a description and the embedded UML diagram. Any author wanting to add or update a design pattern must do the same for the corresponding UML diagram.

The *umple/umple* project is the repository for the Umple programming language.[7] As a model-oriented programming language, Umple heavily relies on UML diagrams, justifying it being the perfect outlier.

[7]See `https://cruise.umple.org/umple/`

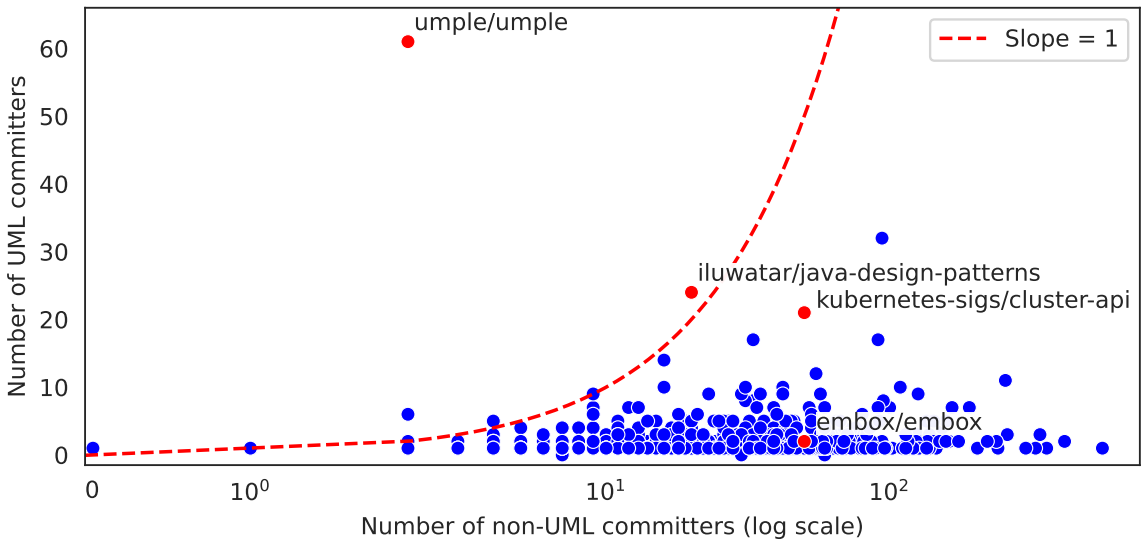


Figure 5.7: Number of UML vs. non-UML committers.

kubernetes-sigs/cluster-api houses a set of APIs for managing Kubernetes clusters, and keeps a Markdown book which documents the APIs[8] (*e.g.,* their implementation, usage, extension). The documentation and proposals for new features contain a wealth of UML diagrams to support the text. The diagramming work is uniformly spread among 28 contributors. Each author makes an average of less than 2 UML commits.

The previous three repositories are in stark contrast with the typical repository in our dataset, for which we consider *embox/embox* as an example. This repository has 154 authors, and only 2 have actively worked on UML diagrams. The top-5 contributors by the number of commits in Table 5.6 include both UML committers. This is closer to what we saw in the contribution periods of the *alsa-project/alsa-lib* and *rolisteam/rolisteam* repositories.

Table 5.6: Top-5 contributors by number of commits for embox/embox. (UML committers)

GitHub Profile Name	Commits	UML Commits
Anton Bondarev	5,581	0
Anton Kozlov	3,154	28
Alex Kalmuk	2,811	0
Denis Deryugin	2,584	0
Eldar Abusalimov	2,252	41

Number of Commits by UML vs. non-UML Committers. We wanted to see whether UML committers contribute more to projects than non-UML committers. We hypothesized that UML committers are more likely to be main contributors: UML committers should be more familiar with the project and more capable of making diagrams; they want developers to be able to contribute easily, so they provide diagrams that help onboard; they want users to use their project and diagrams are a helpful tool to convey how it works.

[8]See `https://cluster-api.sigs.k8s.io/`

In *kubernetes-sigs/cluster-api*, the main contributors were not the main UML diagrammers, as in *embox*. Previously, we analyzed outliers, but UML committers make 4 times more commits than non-UML ones (Mann-Whitney U test $p \leq 0.01$, Cliff's delta $\delta = 0.69$, large effect).

In Figure 5.8, we see a box plot of the average number of commits by UML and non-UML committers. Note that we show the box plot without outliers to focus on the general trend, but we found repositories where UML committers make 534 times more commits. These repositories are similar to *rolisteam/rolisteam*, with a single contributor who makes most commits and is the sole UML committer.

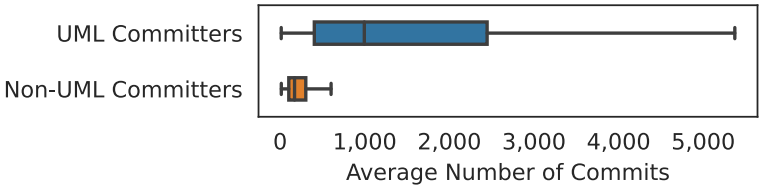


Figure 5.8: Average number of commits by UML vs. non-UML committers.

Dedicated UML Diagrammers. How often are there dedicated UML diagrammers (*i.e.*, those who modify UML diagrams but do not modify source code)? Looking back at Table 5.4 it seems unlikely that the 5.5k commits of Renaud Guezennec are all UML-related. We analyze UML committers who did not modify source code files.

In Table 5.7, we show the 13 repositories we found. Only 2.4% of repositories have dedicated diagrammers. Manually analyzing the repositories in the list more in-depth, we see that almost half of them contain little to no source code (🗎).

Table 5.7: Dedicated diagrammers. (🗎 Little or no source code.)

	Repository Name	GitHub Profile Name	Commits
	adorsys/open-banking-gateway	Dora Nziali	16
	adorsys/xs2a	Daria Lavrenova	50
		Olga Levandovska	16
	apache/isis	Alexander Schwartz	21
	deegree/deegree3	Danilo Bretschneider	26
🗎	*edmcouncil/fibo*	Mike Bennett	85
🗎	*hyperledger/aries-rfcs*	Brent	52
		ashcherbakov	41
		Vinomaster	32
🗎	*kubernetes/enhancements*	Patrick Ohly	61
🗎	*openshift/enhancements*	Enxebre	17
🗎	*progit/progit*	Igor Murzov	156
		Anthony Gaudino	24
	programmevitam/vitam	Clemence Boyer	64
		edith	32
	uportal-project/uportal	Christian Cousquer	21
🗎	*w3f/polkadot-spec*	Fabio Lama	1378
	wix/detox	wixmobile	343

w3f/polkadot-spec is a specification repository, *progit/progit* is a book, *edmcount/fibo* holds a specification for an ontology, and *kubernetes/enhancements*, *hyperledger/aries-rfcs*, and *openshift/enhancements* are repositories used to discuss enhancements or features for other repositories. *wix/detox* is a source code repository but the author who modified only UML diagrams is a bot account for publishing documentation.

One example of a dedicated diagrammer in a repository with source code is *Daria Lavrenova* in the *adorsys/xs2a* repository. Their GitHub profile shows that they are a program manager (PM). Based on their commit history, which is related to documentation and roadmap planning entirely, it looks like they are doing program management work for *adorsys/xs2a*. *Olga Levandovska* has a similar commit history in the same repository and is indicated as PM in their GitHub profile, so we can assume they also have a similar role.

Dora Nziali of *adrosys/open-banking-gateway*, a different repository of the same owner, has a similar commit history to *Daria* and *Olga*, but no reference to PM activities in the GitHub profile. Still, given the similarity, it looks like *adorsys* has PMs dedicated to updating roadmaps and documentation, including UML diagrams.

In our dataset there are only few software systems where people are dedicated to UML diagramming without contributing to source code, this is indeed a rare occurrence in OSS.

Summarizing: RQ_4 focuses on developers who create and maintain UML artifacts. There are very few UML committers, which supports the idea that UML is not used to its full potential. If developers in a collaborative development platform contribute almost exclusively to source code and not to UML diagrams, it might imply that they do not perceive UML diagrams as useful as the source code itself. This is just one of the potential causes and it needs further investigation.

RQ_4 — Findings

- F_1 UML committers tend to be among the longer-standing members of a project.
- F_2 UML committers make ∼4 times more commits.
- F_3 There are almost never dedicated UML diagrammers. Only 2.4% of repositories have contributors who modify UML diagrams and not source code.

5.5 Discussion

The rise and fall of UML cannot be reconstructed by file extensions alone, but through the extensions we could find a reliable source to analyze UML's long-term evolution. UML has never been widely popular in open source software. Our approach led to insights on how the nature of UML support changed from standalone graphical tools, to IDE plugins, to human-readable text. Despite its relative popularity, in this latter form, the graphical quality of the rendered diagram (thus its usability) is still subject to many compromises.

In terms of our considered activity metrics, UML repositories are not significantly different from non-UML ones (except for a slightly higher number of commits). What stands out though is that the majority of them use Java as their main language. We attempted to cluster languages of UML repositories to find commonalities but to no avail. Further research on how UML is used in repositories of different languages might shed light on the features that make UML so appealing for certain languages.

Contributors who modify UML files, on the other hand, are longer-standing and more active members of a project. Although only a small percentage of developers usually contribute to UML, there are exceptions where UML is a collaborative effort (*e.g., kubernetes-sigs/cluster-api*).

Nevertheless, although commits are a coarse-grained unit of measure, the number of UML commits per single author is still minimal (less than 4 in *kubernetes-sigs/cluster-api*).

New tools and human-readable formats should consider several aspects to keep this growing trend of UML adoption in OSS development (*e.g.,* layouts, reducing tool fragmentation, separating presentation and content concerns). Focusing on artifacts partially obscures the motivations behind some of the highlighted phenomena. Nevertheless, making these phenomena intelligible and more transparent for the software engineering audience (especially beyond the modeling community) could spark new studies on the *Whys* driving them.

Underutilization. Going back to our initial meta-question, the underutilization of UML is, in our opinion, tied to the lack of standard unified tools. The software market in the early 2000s generated so many alternatives that the fragmentation created a Babel of UML realization dialects, competing with the "official" XMI specification which, in the end, shared XML's fate. Most commercial tools have expensive licensing options, making them unlikely choices for volunteers. A textual representation created and modified in any text editor increases the *pool* of *potential* contributors. The fact that this *potential* does not result in an *effective* increase indicates that some factors counterbalance this benefit. We argue that one of them is the shortcomings of current tools for human-readable formats (*e.g.,* lack of layouting control).

Reinforcing the economic explanation, there is a focus on *simple* representations without dedicated tooling, where the serialization of the model is the text itself. This agrees with the surveys indicating that UML is more often used with a loose syntax. The goal is not model-driven development, but rather having a "satisficing" (*i.e.,* satisfactory and sufficing) model to support the design phase or to become part of the system's documentation for program comprehension. Nevertheless, in most repositories, only few contributors deal with UML artifacts and those who are not developers are even fewer.

If we do not take into account the shortcomings of the new wave of UML tools and formats, the resurgence wave might not bring UML out of the underutilization condition.

Textual vs. Visual Model. What is given up in the current implementations of text-based UML is the visual power tied to element positioning and layout. We highlighted how core developers usually cover UML creator and maintainer roles. Familiarity with source code may influence their tendency to prefer a textual representation for graphics. Human-readable UML formats do not need to be complicated to be usable for complex systems, but some original features of the visual language must be re-incorporated to maximize effectiveness.

Implicit validation of the model with a non-ambiguous grammar for the language can guarantee syntactically correct models, reducing the burden of correctness for the creator.

Appropriate layout strategies could give back the visual expressiveness and clarity that separates functional from dysfunctional UML diagrams. Finally, we argue that one of the advantages of tools like PlantUML and Mermaid is that they provide a subset of the UML specification in a digestible format. The most useful features are the easiest to use. For example, a class diagram with a few entities is just a few lines of text, moving back the focus on the design and the relationships of composing elements. Automatic layout of elements in a consistent and controlled way will probably be the make or break feature of such tools.

Finally, a reflection on teaching UML. Our study informs and justifies the presentation of currently trending tools (*e.g.,* PlantUML, Mermaid) and encourages shifting the emphasis from the graphical to the conceptual model, distilling it into human-readable textual formats.

5.6 Validity and Limitations

Construct Validity. In RQ_1 we investigate UML extensions as a way to capture UML files. The list of UML extensions cannot be exhaustive. Since the resulting list is used throughout our analyses, we mitigate this risk by integrating multiple strategies to obtain the final dataset of repositories and their UML files: We search for popular UML tools and include their supported input and output formats; we search for examples and counter-examples of UML diagrams by analyzing all UML file extensions in our dataset; we apply progressive refinement of our file tagging strategies for each extension until saturation, to be able to capture all UML files.

In our initial dataset, we filter projects based on criteria that are not directly related to the project's level of documentation. While the attitude towards documentation in general can indicate the possibility to find UML, filtering based on such criteria (*e.g.,* a minimum amount of documentation) would misrepresent the actual popularity of UML.

While developers may use temporary diagrams (*e.g.,* sketches for live discussion), they usually do not persist them. We focus on UML artifacts intended for long-term project support, transient ones are out of the scope of our study.

Internal Validity. Discriminating UML and non-UML content in files can be subjective. The first author collected examples and counter-examples of UML files. Dubious cases were discussed among the other authors until a consensus was reached. To further mitigate this risk, we restricted our analysis to the files that could be described non-ambiguously by automatic UML tagging strategies (*e.g.,* regular expressions for file content to identify specific UML "signatures") or exhaustively manually annotated.

External Validity. We extract our dataset from open source projects hosted on GitHub. This is a threat to the generalizability of our results to other types of projects (*e.g.,* closed source in a company). When creating our dataset, we perform a trade-off to have a large enough sample while mitigating the risk of underestimating the presence of UML due to simple projects. On the other hand, this filtering strengthens our conclusions on the underutilization of UML.

5.7 UML and the Documentation Landscape

UML has been taught as the "be-all and end-all" of software design and modeling to generations of students. On the other hand we find practitioners using small subsets of UML at best, often in an informal, imprecise declination. Only a tiny fraction of relevant GitHub projects include any form of UML diagrams. Something in the chain fell apart between academia and practice. The new wave of interest about UML, represented by the recent resurgence of human-readable text-based formats, should be met with prompt reaction. We need to address the shortcomings we foresee, like the diminished value of a diagram with an uncontrollable layout, to leverage the advantages that textual representations bring, for example, ease of parsability, reviewability, and comparability. The timeliness with which the community will increase its awareness of the implications of the recent evolution will hopefully make a difference in the survivability of UML in its new incarnation.

UML is back and popular again but still underutilized. The lack of maintenance that new formats of UML diagrams seem to experience is nothing new for the research community. What is different is that for these types of artifacts we could find better ways to support design and documentation efforts. UML is a precious resource for large software systems and tool support cannot ignore some of the trends we highlighted, first and foremost, the widening gap between repositories merely containing and those actively modifying UML files. In this regard, standalone tools disconnected from the evolving source code can become more of a hindrance than a help.

Finally, projects with and without UML are almost indistinguishable from the surface, but there is a large gap between contributors who work on UML artifacts and those who do not. The lack of dedicated figures responsible of creating, maintaining, and evolving UML diagrams should make us rethink the role of UML closer to developers than to the mythical figure of the software architect. The alternative is to embrace a no-code philosophy augmented by large language models, dreaming again the 2000s dream of round-trip engineering, and using phone pictures of whiteboard sketches as GPT-Xy prompts asking to produce a text-based UML diagram.

5.8 Summary

Since its inception, UML has been touted as the way to go when it comes to designing and documenting software systems. While being an integral part of many university software engineering programs, UML has found little consideration among developers, especially in open source software. Reasons for this include that UML shares some shortcomings with other forms of documentation (*e.g.,* limited availability, outdatedness, inadequate level of detail).

We presented a study to investigate the evolution and the current situation regarding the use of UML in open source projects. We mined and analyzed ∼13k GitHub projects, developing strategies and heuristics to identify UML files through their extensions and contents, for a quantitative analysis of two decades of evolution of the usage of UML. We explored the popularity of UML, derived characteristics of projects leveraging UML, and analyzed the authors, creators and maintainers of UML artifacts.

Our study confirms that UML is indeed still under-utilized. At the same time we found evidence of a resurgence coinciding with the popularity of human-readable text-based formats, defined and used by tools like PlantUML and Mermaid. We discussed how identifying and addressing the new challenges implied by this resurgence could impact the future of UML as a design and documentation means.

Part III

Reifying the Documentation Landscape

Visualization, Comprehension, Tools, and Case Studies

6

Modeling the Documentation Landscape

The documentation landscape is a snapshot of reachable documentation sources related to a software project. To make this explorable representation useful, we need to add information about the sources. To reify the landscape we need proxy measures of the informative content of a documentation source. As proof-of-concept, we studied an instance of the instant messaging source in the community archetype: Discord.

The last decade has seen the rise of global software community platforms, such as Slack, Gitter, and Discord. They allow developers to discuss implementation issues, report bugs, and, in general, interact with one another. Such real-time communication platforms are thus slowly complementing, if not replacing, more traditional communication channels, such as development mailing lists. Apart from simple text messaging and conference calls, they allow the sharing of any type of content, such as videos, images, and source code. This is turning such platforms into precious information sources when it comes to searching for documentation and understanding design and implementation choices. However, the velocity and volatility of the contents shared and discussed on such platforms, combined with their often informal structure, makes it difficult to grasp and differentiate the relevant pieces of information.

We present a visual analytics approach, supported by a tool named DiscOrDance, which provides numerous custom views to support the understanding of Discord servers in terms of their structure, contents, and community. We illustrate DiscOrDance, using as running example the public Pharo development community Discord Server, which counts, to the date of the study, ∼180k messages shared among ∼2,900 developers, spanning 5 years of history. Based on our analyses, we distill and discuss interesting insights and lessons learned.

6.1 Visualizing Discord

Ever since the advent of internet, digital communities have been born, have thrived, and have also died out. Early platforms were purely text-based (*e.g.,* mailing lists, Internet Relay Chat), while modern platforms, such as Slack[1] and Discord,[2] are full-blown multi-media environments with high velocity and throughput. Global software communities are scattered around the planet and, also driven by the open source movement, have embraced such platforms early on. Each software community uses various communication mechanisms to keep in touch and to discuss [249]. While some of those communication channels can be mined fruitfully [18, 92], the signal-to-noise ratio of certain of those channels is low [108].

[1]See `https://slack.com`
[2]See `https://discord.com`

In the last decade, more feature-rich alternatives have emerged. Rich content media sharing in instant messaging software (*e.g.,* Slack) broadened the spectrum of possible interactions between members of these virtual communities and turned such platforms into precious information sources. Recently, tools originally targeted at video-gaming communities (*e.g.,* Discord in Figure 6.1) have seen an increasing adoption in other contexts, such as classrooms [157] and software developers communities at large.[3]

Developers use these communication channels to promote the libraries/frameworks they developed and offer technical support. Novice developers can ask for help and receive answers from their more experienced peers. In a nutshell, these platforms act as a novel source of documentation and encapsulate design decisions and implementation choices. However, as already pointed out by Jaanu *et al.* [108], the velocity, volatility, and transient nature of the information exchanged on such platforms, combined with their informal structure, makes it difficult to grasp and differentiate the relevant pieces of information.

Our approach aims at easing the comprehension of relevant aspects about the community and its individuals. Apart from the approach and the presentation of DiscOrDance, the main contribution of this paper is a set of custom views to progressively disclose information about:

1. the ***server***, its structure, and its content subdivision;
2. individual ***channels***, their history, and their potential information content;
3. ***authors*** as individual entities, their activity patterns and interactions with the community;
4. ***source code elements*** that can be mined for insights on domain-specific aspects.

6.2 Discord: An Instant Messaging Case Study

We present an instant messaging case study. As we have seen in the related work, Discord is one of the most recent instant messaging platforms adopted by many software development communities.[4] Discord is a *Voice over Internet Protocol*, instant messaging, and digital distribution platform (Figure 6.1). It is a client/server application with support for peer-to-peer communication. A Discord server is the basic functional unit representing a community. Figure 6.1 shows the user interface (UI) of the Discord desktop application. Discord is available for desktop (*i.e.,* Linux, Windows, macOS), tablets, and smartphones (*i.e.,* Android and iOS).

The importance of software community platforms is increasing and is fundamentally changing how developers discuss and interact with each other. What sets instant messaging apart is the throughput of the channel and the volatility of its content.

Aside from DiscOrDance, no other tool allows to grasp the structure of a Discord server and to retrieve specific information about software (*e.g.,* source code blocks). Discord supports word-based search in the whole history of a channel, with filtering criteria such as date, sender, and presence of embedded links or images. Although this might work for a quick search in a low-traffic channel, it does not scale up to generic information retrieval on large servers. Dealing with high throughput is essentially left to manual labor by users. This makes Discord the ideal candidate for a case study on retrieving content for the documentation landscape.

[3]See `https://git.io/JnRGr`
[4]See `https://tinyurl.com/dev-discord-top-10`

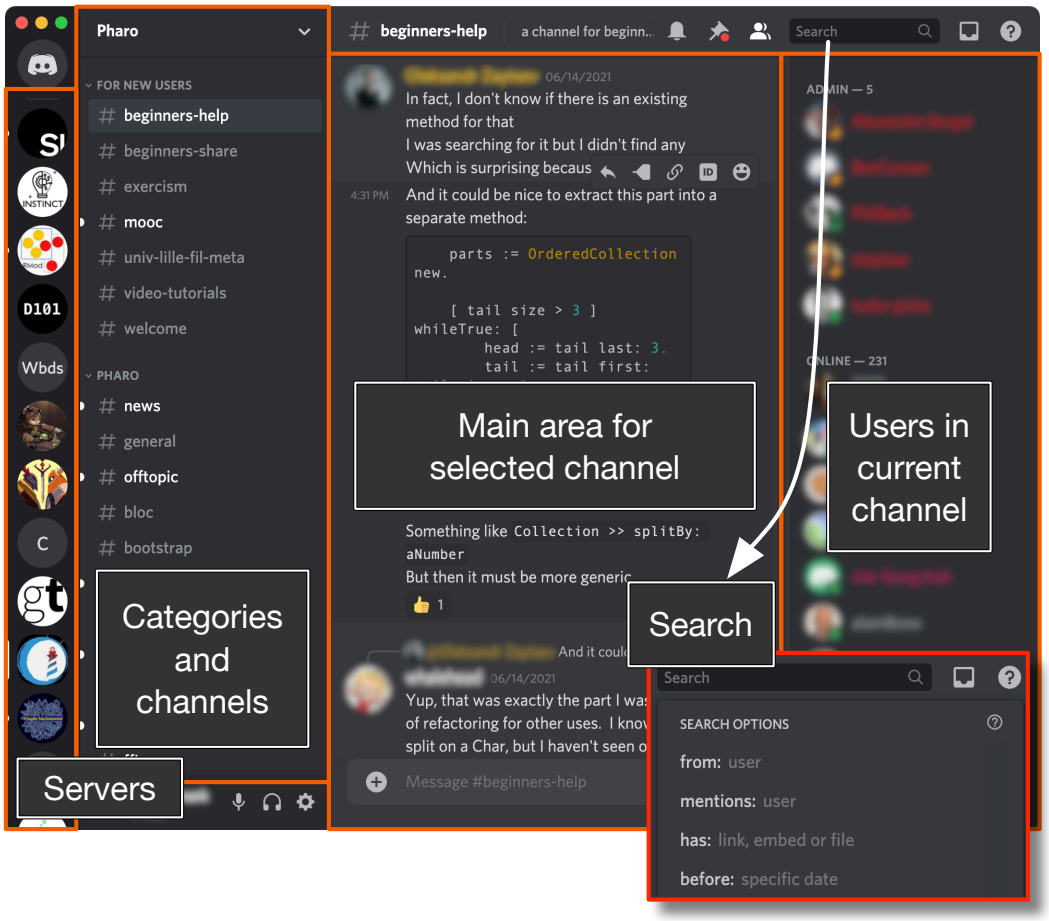


Figure 6.1: Discord application: Screenshot of the desktop version.

6.2.1 Discord Servers in a Nutshell

A server is typically divided into ***Categories*** and ***Channels*** (Figure 6.2).

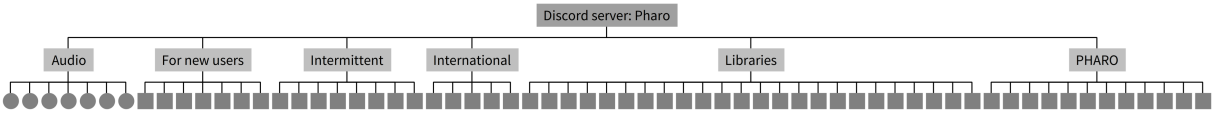


Figure 6.2: Structure of the Pharo Discord server (root): Categories (rectangles), voice channels (circles), and text channels (squares).

The two main channel types are Text and Voice. Text channels support textual messages, embedded links (*i.e.,* textual messages with a partial preview of the linked resource), emojis, reactions, and file sharing. Voice channels support spoken communication, camera feeds, and screen sharing. To better structure a Server, Channels can be grouped into Categories. Discord uses a ***permission system*** based on ***roles*** assigned to the members to limit (or grant) the visibility of a given channel (or a category) to a given role. Members of a server interact with each other in the channels they have access to.

In a server, there are two types of users: Regular (*i.e.,* humans) and Bots (*i.e.,* software applications that run specific activities in a channel, *e.g.,* moderation).

Our supporting tool, called DiscOrDance, features a bot that can be added to a server to retrieve and analyze its data (*e.g.,* messages it is entitled to read). DiscOrDance also uses DiscordST[5] [41], a client for the public Discord REST API written in Pharo.[6] After creating a domain model by scraping the Discord server, DiscOrDance enables its interactive visualization based on the views we present in Section 6.3.

6.2.2 Case Study: The Pharo Discord Server

We analyze the Pharo development Discord server. Pharo is a pure object-oriented programming language and a powerful environment, focused on simplicity and immediate feedback, inspired by Smalltalk. Table 6.1 provides statistics about this server, containing several hundreds of people with an average of 100 messages per day.

Table 6.1: Statistics on the Pharo development Discord server.

Snapshot Date	Jun 16 2021
First Message Date	Sep 8 2016
Activity Duration	4 years 282 days
# Active Members	966
# Inactive Members	1,525
# Previously Active Authors	394
# Sent Messages	183,481

6.3 Visualizing Discord with DiscOrDance

DiscOrDance offers six polymetric views [131] to analyze different aspects of a Discord server such as channels, authors, and source code elements discussed in messages.

6.3.1 Channel Activity View

This view provides an overview of the channels in terms of their activity, *i.e.,* the number of messages sent. Each text channel is represented as a rectangle: The height is proportional to the number of messages sent to that channel while the width is proportional to the number of authors who sent them. The area of rectangles indicates the "activity" in a given text channel. Voice channels are represented as fixed-size circles. Since channel names are not unique, to distinguish them, we keep track of the channel hierarchy (*i.e.,* categories containing them). For this reason, the Channel Activity View adopts a tree layout, with categories at the top.

Examples

Figure 6.3 depicts the Channel Activity View for the Pharo Development Discord Server. At a glance, we can spot the two most active text channels: *"general"* in the *"PHARO"* category and *"beginner-help"* in the *"For new users"* category. The former counts 49,872 messages from 862 authors while the latter counts 23,305 messages from 494 authors.

[5]See `https://git.io/JnR3h`
[6]See `https://pharo.org/`

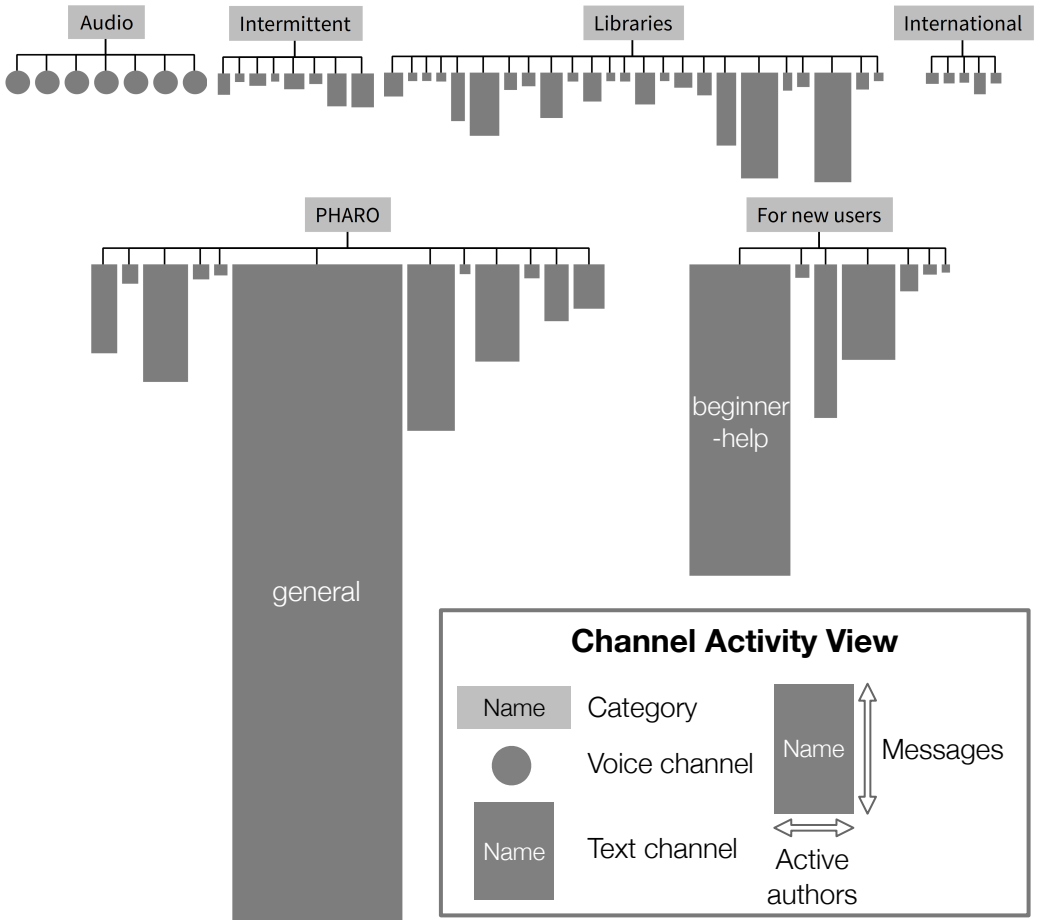


Figure 6.3: Channel activity for the Pharo development Discord server.

6.3.2 Channel Activity Timeline View

Channels can also be analyzed in terms of their recent activity and overall lifespan. When considering the first message sent in a channel as the starting point and the last one as the ending point, we can see interesting patterns.

Examples

The server started as a single channel for a few months. Most channels have recent activity and a long history (Figure 6.4). There are important channels that are still relevant. Their initial activity date and the overall height of the channel's representation can indicate how important the channel has been in the history of the community.

In this view, we see channels that were active in the past and are not active anymore and channels created only recently. The former are candidates for archival and probably do not serve any real purpose besides documenting the past. The latter should be investigated separately since their overall activity may be overshadowed by longer standing channels, despite possibly containing interesting insights or patterns.

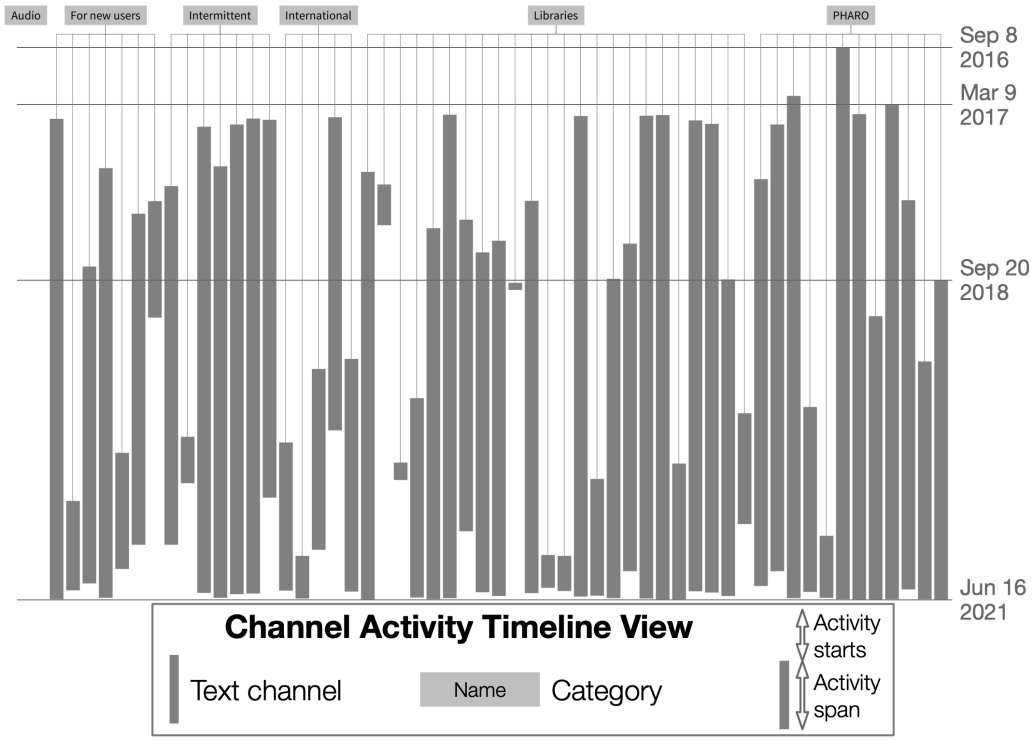


Figure 6.4: Channel activity timeline for the Pharo development Discord server.

6.3.3 Author Activity Status View

A Discord server is dynamic in terms of members and their activity status. Some authors send a few messages and then quit the community while others only read messages without sending anything. This view aims at highlighting the composition of the user base of a server. Table 6.2 summarizes author types and membership status.

Table 6.2: Author types based on activity and membership status.

Activity/Membership	Definition
active member	sent messages, currently receiving messages
inactive member	receives messages (and possibly reads them), didn't send any message (yet)
active ex-member	sent messages in the past, not part of the Pharo Discord community anymore
inactive ex-member	never sent any message, presence on the server can be inferred by at least one mention

Examples

Figure 6.5 depicts all authors, sorted by decreasing number of messages, colored by their activity and membership status. There are 966 active authors who are also current members of the community. 1,525 members did not post a message yet. Previously, 394 authors posted at least one message but they left the server, thus, they are not members anymore. Moreover, this view highlights differences in author's behavior. For example, Author 1 (*i.e.,* first row, first rectangle) is more active than Author 2 (*i.e.,* first row, second rectangle) but in a lower number of channels.

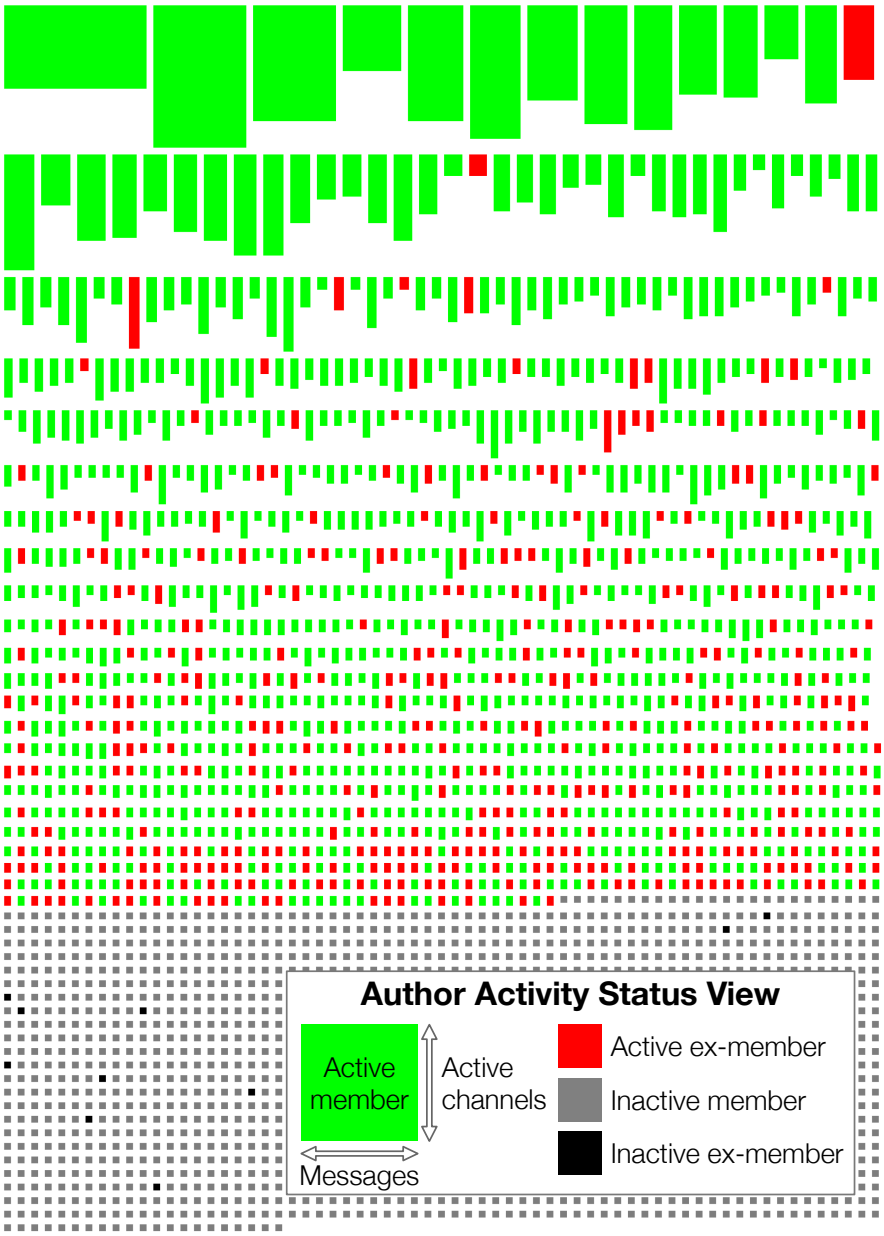


Figure 6.5: Author activity status for the Pharo development Discord server.

It would be interesting to investigate the activities of active ex-members to find insights on why they left the community (*e.g.*, they did not get an answer to their first question, they had a flame with another user, they changed topics).

6.3.4 Author Activity Spark-line View

This is a chart-based view. Every author can have different activity patterns when using Discord to communicate. Activity charts are a compact representation of daily activity by an author. In the single view (Figure 6.6), only one author is considered and his daily number of sent messages can be charted over his activity period or the whole server activity period. Using a small multiples approach [257], we can compare different authors to spot differences in their activity patterns.

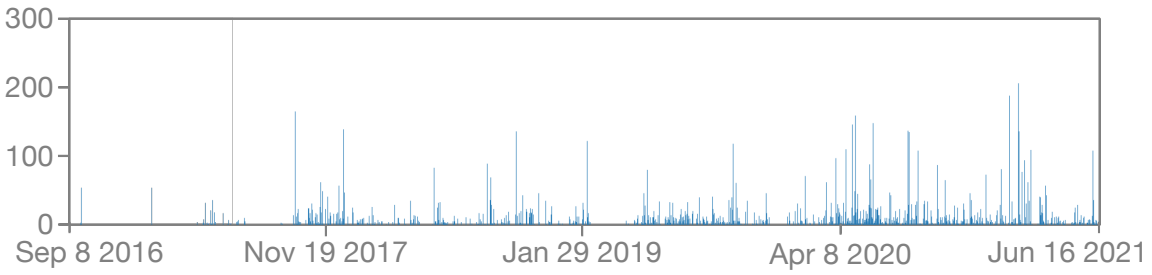


Figure 6.6: Author activity spark-line for Author 9.

Examples

In Figure 6.6, we depict the activity of a long-standing member of the community. The increase in average activity in the last 1.5 years is apparent as well as a certain periodicity in the overall activity that could be further investigated (*e.g.,* with respect to seasonality). In Figure 6.7, we show the top-10 most active authors with their daily activity over the whole server lifetime.

Figure 6.7: Author activity spark-lines for the 10 most active authors.

Authors 1, 2, and 3 (from top-left by row) are still active while 4, 5, and 6 have stopped their activity between around 1.5 and 2.5 years ago. Author 7 is the most active, while his activity started more recently compared to the others. Author 10 has a very low average daily activity.

6.3.5 Code Blocks View

Many messages feature structured content of various type, such as stack traces and diffs. We are interested in source code, which developers frequently share and discuss using Discord. The Pharo Discord server features close to 14k messages with structured content, and more than 2.3k messages with source code. Discord supports Markdown that allows marking code blocks in a message, for example, with the language to use for highlighting keywords in a snippet of code.

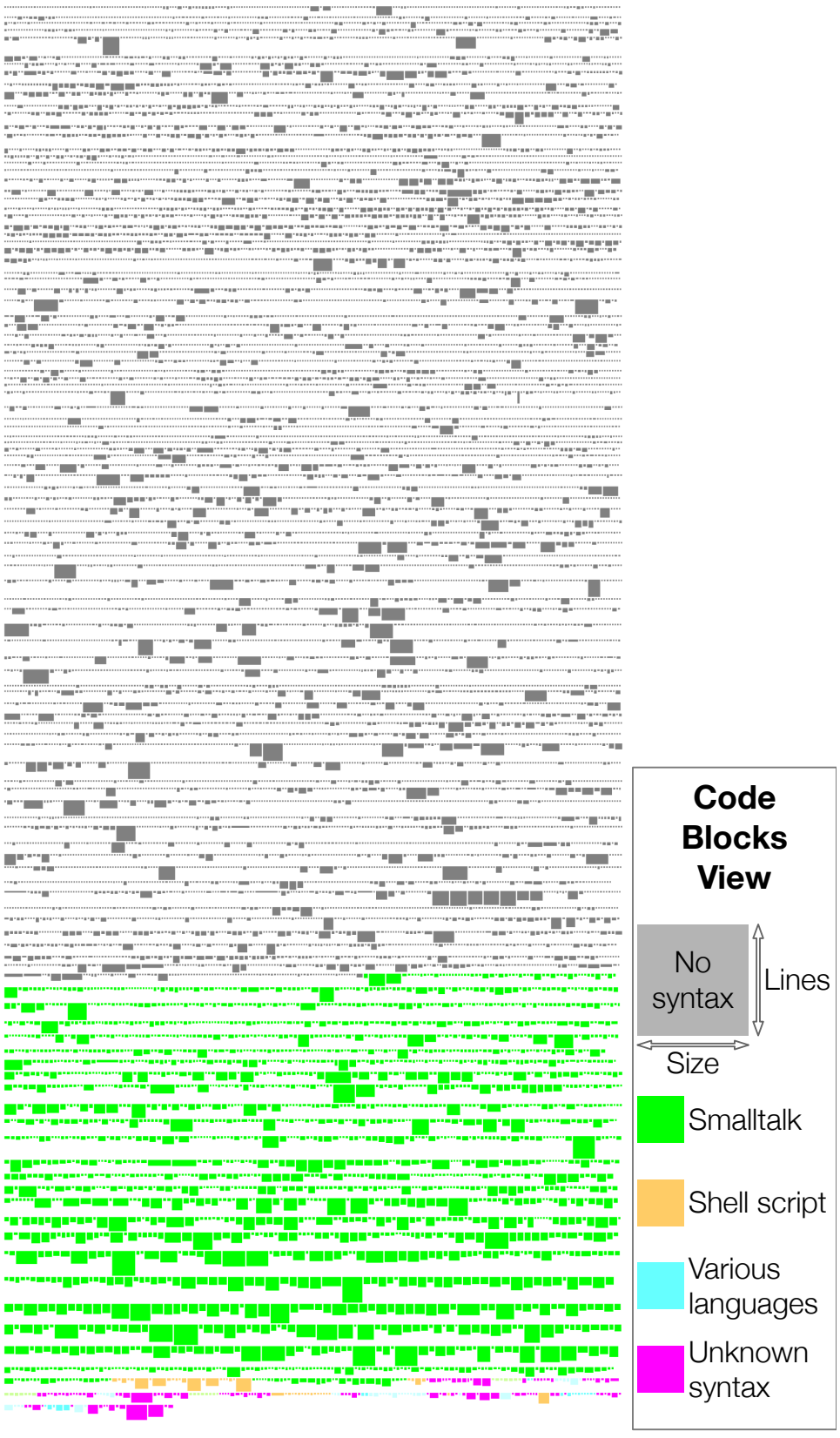


Figure 6.8: Code blocks for the Pharo development Discord server.

We use the syntax highlighting annotation (*i.e.*, ‘‘‘smalltalk) to identify the programming language of a code block, as shown in the following sample message:

````
‘‘‘smalltalk
    MyClass new doThing: (MyClass new doAnotherThing)
‘‘‘
is equivalent to:
‘‘‘smalltalk
    [ :myClass | myClass doThing: myClass doAnotherThing ] value: MyClass new.
‘‘‘
````

Examples

In Figure 6.8, we show the potential code blocks, using specific colors for those with a recognized syntax highlighting. The vast majority of source code elements are 2,530 Smalltalk code blocks.

6.3.6 Class References View

Narrowing down the presence and relevance of source code related information in our case study, this view investigates the number of mentions for specific classes in the Pharo core libraries. We restrict the code blocks to the ones explicitly marked for Smalltalk syntax highlighting. We then perform a regular expression based pattern matching with class names to extract class mentions.

Examples

In Figure 6.9, we show mentions of the Collection hierarchy. We sort them by number of mentions, highlighting the most common classes in Smalltalk code blocks. There are 294 references for the *String* class, 212 for the *Dictionary* class, 185 for the *OrderedCollection* class, etc.

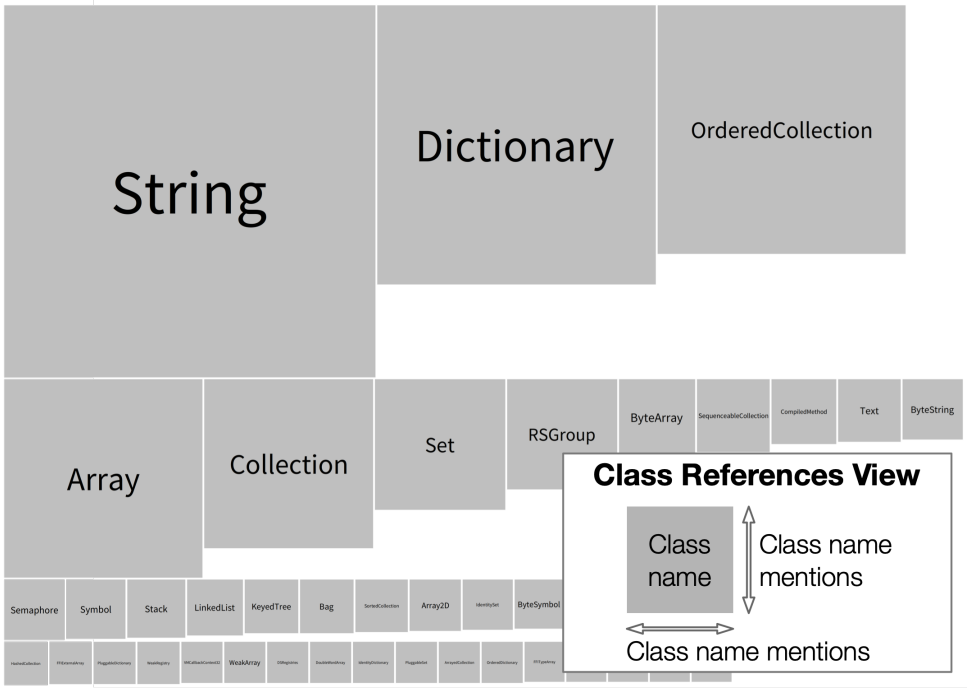


Figure 6.9: Class references for the Smalltalk Collection hierarchy.

6.4 Summary

The importance of software community platforms is increasing and is fundamentally changing how developers discuss and interact with each other. We presented a set of views, generated using a custom-built tool named DiscOrDance, exploring one instance of such a platform: Discord. Beyond the views that we presented, and the insights that said views allow, our primary contribution is a comprehensive approach, preceded by a careful modeling of the domain, to visually navigate and explore novel types of information, that ultimately all relate to software systems, and their understanding from the point of view of the developers discussing them.

Discord is a treasure trove of information and even a relatively small server, like that of our case study,[7] can be mined to find hundreds of code snippets that can contribute to program comprehension as usage examples or other types of documentation.

These snippets do not live in isolation. They come with entire conversations surrounding them. In these conversations, text in natural language explains, investigates, fixes, and in general discusses about these code snippets, providing valuable information that, at the moment, is accessible only to the members of the specific communities.

In the next chapter we will see how to overcome a fundamental limitation of investigating these sources with *their* fundamental unit of measure: A single message. With conversations promoted to first class citizens we abstract and allow a higher level representation of the contents of a Discord server.

[7] Small if compared to communities with hundreds of thousands of users and throughputs which are orders of magnitude larger than in the Pharo server (*e.g.,* the Reactiflux Discord server).

7

Using Discord Conversations as Program Comprehension Aid

Modern communication platforms used in software development host daily conversations among developers and users about a wide range of topics pertaining to software systems, such as language features, APIs, code artifacts like classes and methods, design patterns, usage examples, code reviews, bug reporting and fixing. Discord servers are one of these virtual community hubs that have seen a steep rise in popularity, as coordination and aggregation means for communities of developers. Although Discord supports filter-based search functionalities, the sheer volume, velocity, and small granularity of single messages make it hard to find useful results, let alone complete discussions revolving around particular themes. One reason is that the concept of a discussion, which we call a *conversation*, does not exist as an explicit concept. We argue that extracting and analyzing such conversations can be used fruitfully to aid program comprehension.

We present an approach that reconstructs the conversations that take place on a software community Discord server, focusing on software-related conversations: Our approach binds the conversations to the discussed artifacts. Leveraging our approach, we built a tool that enables the interactive exploration of the conversations' contents. We illustrate its usefulness through a number of examples that highlight how the insights obtained serve as an additional form of software documentation and program comprehension aid.

7.1 Conversations and Understanding

More than half a century ago, McLuhan's seminal book "Understanding Media" started on the premise *"the medium is the message"*: A communication medium itself, and not [only] the messages it carries, should be the primary focus of study [154].

Indeed, the very nature of a communication medium shapes the contents it supports. In recent years the many types of media that have been used by software communities have been complemented, if not replaced, by a new one: rich-media instant messaging platforms, such as Slack and Discord. These platforms are high-throughput/high-volatility virtual hubs where developers discuss daily about software. Gitter, Slack, and Discord are currently some of the most used instant messaging platforms, and have also been studied as possible data mining sources for software-related information [43, 66, 144, 175, 181, 225, 241, 244]. Despite their popularity, they all lack semantic-based searches in the documentation corpora they provide. They offer at most basic functionalities to group messages in semantically coherent chunks, such as Q&A threads. The basic functional unit of these platforms is the single message.

The message granularity level is too fine-grained to efficiently help a developer quickly discriminate between the content of interest and noise. Searching answers for a specific task amounts to defining filtering criteria (*e.g.,* keywords or date intervals), retrieving messages, and manually exploring them one by one. This puts the burden of even finding the limits of the region of interest on the developer, without any form of summary to speed up the discrimination process in order to find what is needed to accomplish a task. Many GitHub projects adopted Discord as the primary communication tool among development team members and their community. Unsurprisingly, concerns about the long-term persistency of information shared on Discord are already emerging among its users:

> *"I can't wait for the day Discord starts to cull old content [from their content delivery networks] to save server space and for so much information to just disappear.*[1]"

Developers use these platforms to share knowledge and coordinate projects [108]. This integral part of the documentation landscape [180] constitutes an important form of crowd-sourced documentation for many languages, frameworks, and software projects [51]. Besides preserving history, an imminent need is to access and mine this documentation source. The volatility of information is in the very nature of instant messaging applications, for example, in the fact that only a handful of messages is visible on the screen simultaneously, and that older ones quickly disappear from a user's point of view. This can be a matter of seconds or minutes in a high-traffic channel. Countering the volatility could help in the fruition of information in real-time.

We propose an approach to mine knowledge in Discord servers' crowd-sourced documentation to aid in program comprehension, while providing an extra layer of persistence. We reconstruct conversations and elevate them to first-class concepts. We show how discussed source code artifacts can be analyzed to extract knowledge about them as a form of ad-hoc documentation. We provide a visual representation of conversations that can be leveraged as a form of summarization to gauge important aspects that could play a role in exploration strategies. Finally, we outline the links between source code and the natural language part of a conversation.

7.2 Discord Conversations

Discord is a rich-media instant messaging, *Voice over Internet Protocol* (*i.e.,* VoIP), and digital distribution platform. Users can communicate with messages containing text, images, videos, files, embedded links, and emojis. It also supports streaming, voice chat, and video conferencing.

A *Discord server* is the basic functional unit encapsulating the concept of a community. Users in a Discord server can share messages in *text channels* and talk in *voice channels*. Channels are organized into *categories*. Many software development communities have a public Discord server, with a permanent invite link (*i.e.,* published on the main website or GitHub project page) that allows users to join the server and participate in activities.

Problem – Discord servers can host tens of thousands of users and reach throughputs of several messages per second.[2] Nevertheless, the Discord client for desktop can fit up to twenty one-line messages on an average screen, which drops to just a handful on mobile devices. People can easily miss longer conversations while they are offline: Why and when did a conversation start? Hard to tell at a glance. One needs to scroll to see all messages, and maybe realize that nothing important happened.

[1] See `https://knockout.chat/thread/33251/1#post-1176126`

[2] The Programmer's Hangout Discord server has more than 110,000 users with ca. 17,000 active users a day. See `https://disboard.org/server/244230771232079873`

Solution – Reconstructing summaries of conversations to show appropriately chunked pieces of information to users. The first step in this direction is to aggregate messages and reconstruct conversations, adding meaningful information about the content (*i.e.,* discussed topics) and its context (*e.g.,* involved authors, conversation length). This higher level representation can help discriminate conversations of interest and easily overview their messages. It can also be used for archival and retrieval.

Case Study – We demonstrate our approach and present interesting insights that emerged from the analysis of the Pharo Discord server, the main communication hub for daily interactions of the Pharo[3] developer community. It has more than five years of history with ∼200k messages and ∼1.6k message authors (see Table 7.1). The server has 67 channels organized into seven categories. The following section presents examples from this server, including conversations on the *roassal* channel in the *LIBRARIES* category.[4]

Table 7.1: Statistics on the Pharo Discord server.

Snapshot Date	Feb 7 2022
Activity Span	5 years 153 days
# Sent Messages	197,009
# Members	3,176
# Active Authors	1,568

7.3 Disentangling Conversations

In most instant messaging applications, including Discord, the minimum unit of exchanged information is a message. We group messages that are temporally related to one another into **conversations**. The boundaries of a conversation can be defined in various ways. In our study, we use inter-message time intervals. Figure 7.1 shows these intervals on the Pharo Discord server.

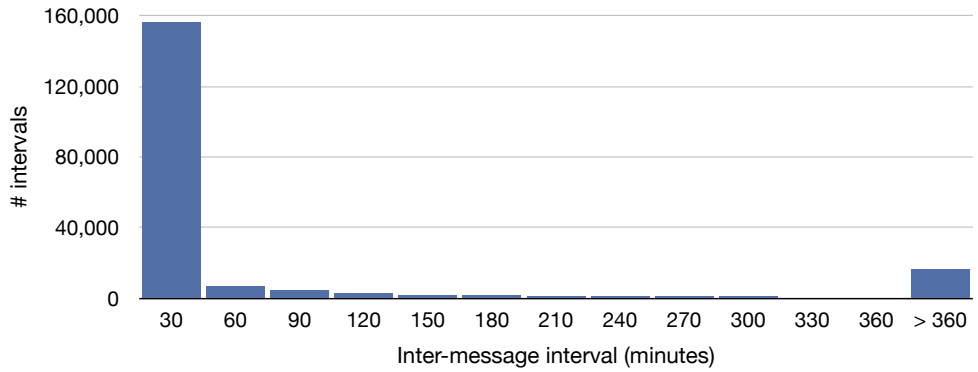


Figure 7.1: Time intervals between messages in all channels.

We use a two hours threshold, retaining 87% of the intervals, to reconstruct *quasi-real-time conversations*. This provides more context, so a human observer could still disambiguate and manually split without losing potentially related information if needed. The impact of the chosen threshold on the accuracy of recommended conversations about a topic remains to be evaluated.

[3]See `https://pharo.org`

[4]Roassal is an agile interactive visualization framework for Pharo [29].

On the Pharo Discord server, we found 26,306 conversations with an average duration of 49.8 minutes and 7.5 messages (see Table 7.2). Considering those with more than one message (16,482), we have an average of 11.3 messages per conversation.

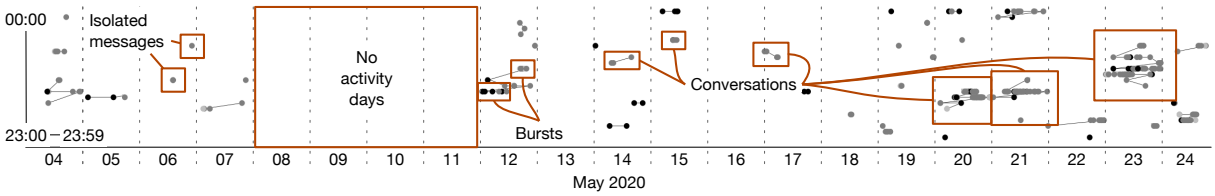
Table 7.2: Conversations on the Pharo Discord server.

# Conversations	26,306
Average Conversation Span	49.8 minutes
Average Messages per Conversation	7.5
Longest Conversation # Messages	532

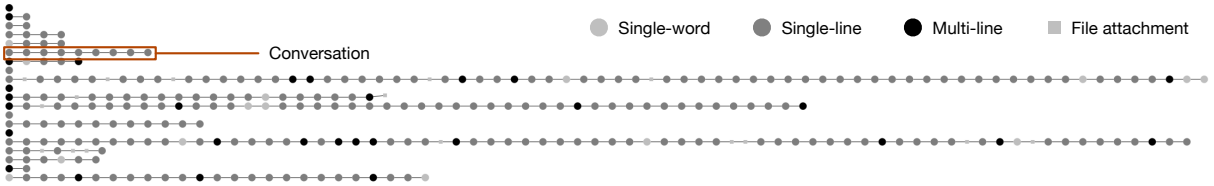
Messages Timeline and Conversation Patterns – Conversations can follow one another, they sometimes consist of sporadic isolated messages, exist as short intensive bursts or longer discussions, possibly with pauses between bursts. We can see such patterns in Figure 7.2a, where messages are placed in a 2D space according to their timestamp. Each day is a column in the view from left to right and each row represents an hour of the day. Messages are represented as dots and connected when they fall within the two-hour threshold. It is easy to spot clusters of overlapping messages, which represent quasi-real-time conversations, with seconds or minutes between them. When messages are linked over longer distances, they indicate a semi-synchronous interaction, *i.e.,* when the inter-message interval is still below the threshold. Horizontal gaps show inactivity periods.

Linearized Conversations and Message Types – A text message can be a single emoji, a word, or sentences on multiple lines. We represent the conversations as a linear sequence of messages to provide a clearer view of the types that alternate in a typical flow.

In Figure 7.2b, we show examples of alternation patterns in conversation sequences of varying lengths. 89.5% of messages (176,356) are composed of a single line. This majority of one-line messages is not surprising if we consider the typical interaction mode of Discord. On a newline keystroke (*i.e.,* Return) a message is sent. Only a specific combination (*i.e.,* Shift-Return) generates a multi-line message. While the average conversation length is low (7.5 messages), there are many significant outliers (SD 19.7). For example, the longest conversation counts 532 messages. Four authors help each other with coding exercises and repository management issues. Further work is needed to ensure that conversations in low-traffic channels are not split.



(a) Partial message timeline where days are represented as columns and rows represent hours of the day.



(b) Linearized conversations and message types (May 19–24, 2020).

Figure 7.2: Two visualizations of message sequences in conversations for the *roassal* channel.

For example, timezone differences may result in responses exceeding the given threshold. In high-traffic channels, disentanglement of interleaving messages should provide better accuracy in reconstructing minimal subsets about the same topic [43]. Isolation or longer intervals between messages could indicate questions that did not receive a timely answer. These occurrences should be investigated separately. Our main focus is on (quasi-)real-time conversations and the source code they discuss.

7.4 Conversations About Source Code

Figure 7.3 shows an example of a conversation between two authors involving source code (fair usage consent has been explicitly granted by authors whose real names or pictures appear in the following examples). There are 1,485 conversations on the Pharo Discord server about source code artifacts. While this specific example is short, interestingly, the average number of messages per conversation containing code is 29, about four times the overall average (7.5). This indicates more activity around source code in the Pharo Discord server.

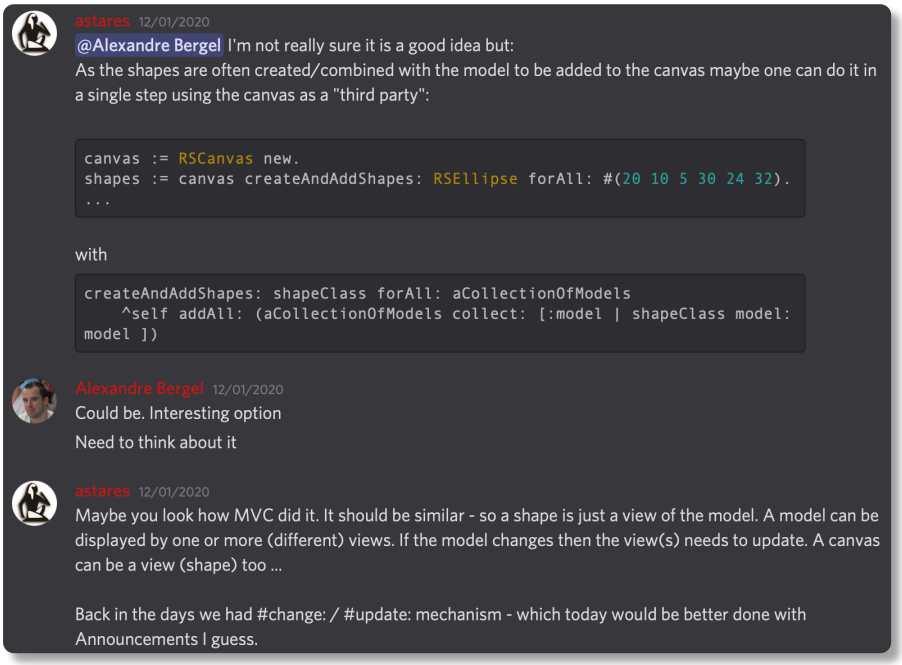


Figure 7.3: Example of a conversation with source code.

To investigate discussions revolving around source code snippets, we developed a custom view that shows the relevant contextual information in a condensed representation. Figure 7.4 shows a conversation consisting of 19 messages between three authors.

Serge Stinckwich sends a single comment suggesting what could be implemented. The conversation revolves around four code snippets (*i.e.,* green rectangles in the center). We highlighted one to show its content in a tooltip. The discussion focuses on how classes could be used to create charts: *RSChart*, *RSScatterPlot*, and *RSLinePlot* (*i.e.,* outer circle). Various methods are also visible (*i.e.,* inner circle). In particular *addPlot:* and *addDecoration:*, two methods of *RSChart*. The visualization shows how this information can grasp the topics of a conversation. This work should be extended to automate information extraction and provide a compact, meaningful representation, *e.g.,* with text summarization.

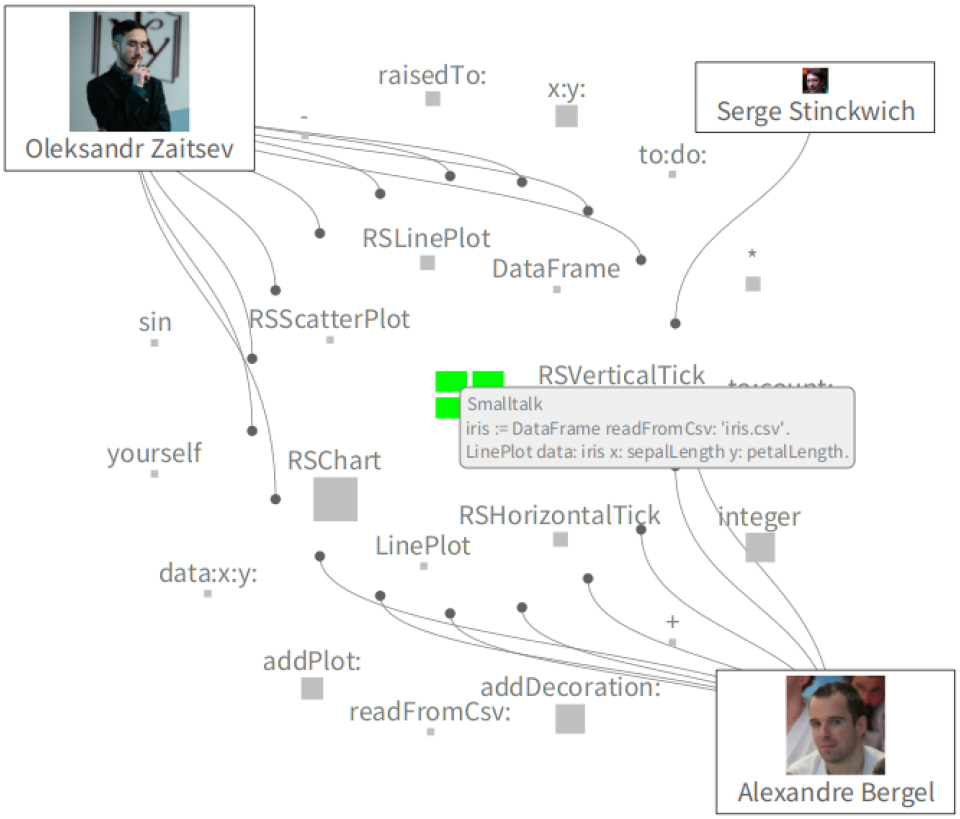


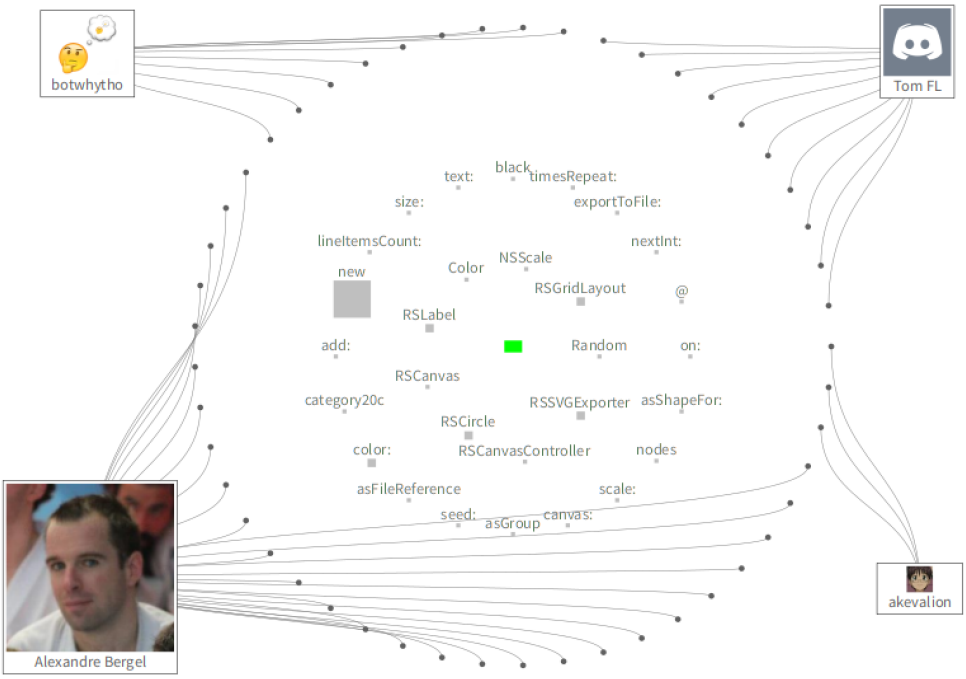
Figure 7.4: Complex representation of a conversation with authors, messages, code, referenced classes, and methods.

How? – We differentiate natural language and source code based on the code blocks in the messages, as they can be marked by single or triple back-ticks, like in Markdown.

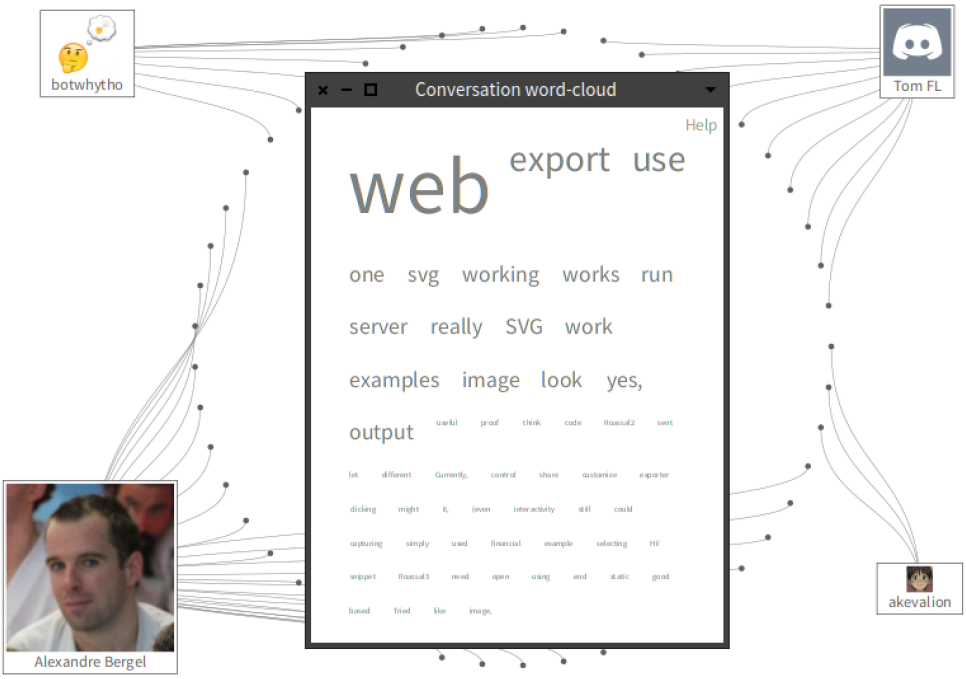
In most cases, the language of a code block is specified for the syntax highlighting, and we rely on it to extract the relevant source code elements and tokenize the extracted code blocks.

In the case of Smalltalk, we do a full parsing to get an abstract syntax tree. For other languages, we implemented a fallback solution based on *ANTLR* parsers. When we cannot parse the source according to a given grammar, we rely only on the tokens gathered through tokenization. Finally, we use heuristics, for example capitalized first letter, to extract class names from tokens (Figure 7.5a).

Why? – Our main focus is to aid program comprehension by retrieving conversations about specific language features, APIs, constructs, methods, and classes. Splitting natural language and code allows us to treat the two differently. While for the natural language content we provide word clouds to grasp the most important terms in the conversation (Figure 7.5b), source code classes and methods are mapped to the relevant conversations. We can retrieve, for example, all the conversations containing code about the *RSSVGExporter* class and find the conversation in Figure 7.5 whose main topic is about *exporting* the Roassal canvas to be *used* in the *web* (top-keywords in the word-cloud).



(a) Code tokens.



(b) Natural language word-cloud.

Figure 7.5: Example of a conversation between four authors with messages, source code and natural language related features.

7.5 Advanced Disentanglement

Instant Messaging (IM) applications, such as WhatsApp, Slack, and Discord, are ubiquitous, supplanting asynchronous communication means (*e.g.,* emails) for professional and personal use. Developers are no exception: Software communities are born, evolve, thrive, and sometimes die on such virtual communication hubs. This paradigm shift also induces concerns about information persistence and accessibility. IM platforms from private companies (*e.g.,* Slack, Discord) usually provide an API to access information on public servers. However, this access is often restricted (*e.g.,* Slack's free tier plan provides access only to messages of the last 90 days) without guarantees on future availability of public content.

When a community has a wide scope or simply grows in popularity, scalability problems arise. Many public servers feature channels where questions and answers about a specific topic can be sent. High-traffic channels experience very soon an interleaving of messages pertaining to different conversations (interleaving colors, Figure 7.6 left), with multiple messages being sent almost simultaneously. A human reader is moderately capable of reconstructing in real-time the conversation flow. Aids like replies indication, temporal intervals, cue words and explicit Q&A structures can be used for disambiguation. However, *reliable automatic reconstruction of conversations remains an open challenge.*

Elsner and Charniak proposed a disentanglement algorithm for IRC [69]. While IRC messages are only pure text, modern rich-media platforms support features (*e.g.,* multimedia sharing, explicit replies, embedded link previews) that can be leveraged for disentanglement. Their algorithm was adapted by Chatterjee *et al.* [42] to disentangle developer conversations in Slack, while Subash *et al.* [244] used it for Discord.

No approach for conversation disentanglement is available out-of-the-box, reducing its usefulness. Having conversations as higher-order constructs provides richer semantics than sequences of messages, and is key to improve the quality of inputs to research on developers' chats.

We present CoDi, an extensible service-based API and web interface (Figure 7.6) for disentangling developer chats. It can be integrated in preprocessing pipelines to clean up collected data. CoDi's input/output formats improve interchangeability of disentanglement algorithms. This is a step towards easier comparison of alternative approaches. The CoDi web interface helps in exploratory phases with fast iteration cycles. The proposed visualization of disentangled conversations provides qualitative (*e.g.,* message grouping) and quantitative information (*e.g.,* accuracy, F-score, computation time) about the disentanglement process with the selected model. The tool and the approach we propose reduce inconsistencies in configuration, simplify the setup, and improve reliability of results.

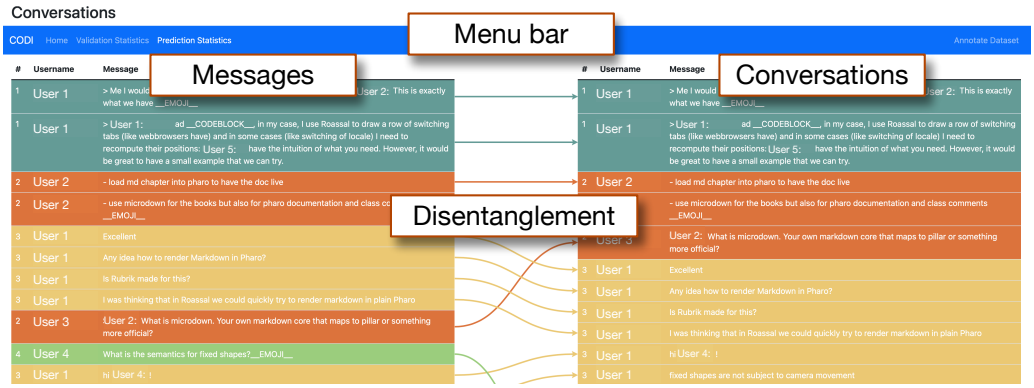


Figure 7.6: Conversation disentanglement interface in CoDi.

7.5.1 Intermezzo: Conversation Disentanglement and Instant Messaging

While most of the related works focus on IM applications for developer communications, few of them properly account for the disentanglement problem in their studies. Instead of semantically related discussion threads, our work on Discord conversations [183] considers (quasi-)real-time interactions. Conversations lasting over a day are still meaningful, but they are handled better by asynchronous media (*e.g.,* mailing lists, Q&A websites, forums), which have been extensively studied, for example by Abreu and Premraj [1], Bacchelli *et al.* [18], Guzzi *et al.* [92], and Di Sorbo *et al.* [57] for e-mails, Parnin *et al.* [173] and Ponzanelli *et al.* [176] for StackOverflow, and Di Sorbo *et al.* [58] for mobile app reviews. This studies dealt with similar issues in extracting information from developer communications.

Conversation disentanglement has been addressed both with unsupervised [2, 150, 222] and supervised [42, 69, 140, 268] approaches. According to Liu *et al.* [150], an orthogonal distinction in disentanglement techniques is between two-step approaches and end-to-end ones. The former [42, 69, 155] combine local relatedness (*e.g.,* message pairs relatedness) and conversation clustering, while the latter [114, 149, 267] try to capture global properties of conversations in one step.

Only a few published datasets have a manually annotated ground truth [69, 129, 155] and of these, only one of them is considered large [129]. Due to this lack, three large datasets [42, 175, 244] of automatically disentangled developer conversations have been proposed for SE research.

The different velocity, granularity, and features of modern instant messaging platforms require building higher level, source-independent concepts. We try to go beyond the limit of message-level granularity by exploring state of the art disentanglement algorithms, like, for example, the work of Elsner and Charniak [69] applied to a new media (*e.g.,* Discord) [189].

7.6 CoDi

CoDi is implemented in *Python* and it is composed of three main modules (Figure 7.7): The conversation *disentangler*, the *REST*ful *API*, and the web-based frontend *client*.

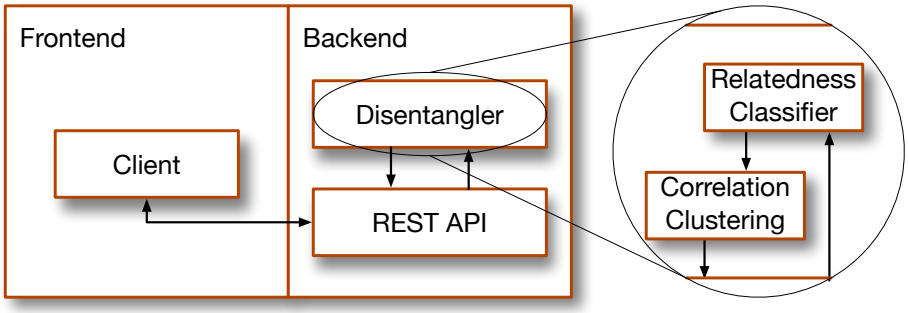


Figure 7.7: CoDi architecture overview.

The disentangler module is a reimplementation of the model proposed by Elsner and Charniak [69] and later modified by Chatterjee *et al.* [42]. Message pairs are evaluated with respect to their relatedness, according to features like time interval and term similarity, and then clustered in conversations based on their relatedness scores. We extend some of the features used by the latest version of the model (*e.g.,* by adding variations of possible greetings, with different types of mention features) and allow to use two different *relatedness classifiers*: Random forest [42] and logistic regression [69].

CoDi's RESTful API provides an interface to access the disentangler by defining endpoints for training, validation, and prediction (Table 7.3). Two endpoints provide utilities to convert input files between different formats and retrieve performance statistics of the model computations (*e.g.,* accuracy, precision, recall, F-score, execution time).

Table 7.3: REST API endpoints.

Path	Type	Description
api/train	POST	Sends an input dataset to the disentangler to train the classifier.
api/validate	POST	Sends an input annotated dataset to the disentangler. It will predict the conversations and compare them to the provided annotations.
api/predict	POST	Sends an input dataset to the disentangler to predict conversations.
api/statistics	GET	Retrieves the disentangled conversations and performance statistics (after *validate* or *predict*).
api/convert	POST	Converts a dataset from *ANNOT* to *JSON* format.

7.6.1 Towards Research Code as Infrastructure

CoDi is a step towards *Research Code as Infrastructure*. This approach aims to improve reliability of results, reusability of implementations, and comparability of approaches in research prototypes for conversation disentanglement. Trying to replicate previous studies [42, 68, 69, 244], we set up an environment with different Python versions, complying with needs of older and newer scripts. We had to include a pre-compiled version of the MEGAM[5] max entropy classifier, a dependency whose latest version dates back to October 2007.

These details took significantly more effort than a file drag-and-drop in a web page and browsing results with performance information of the algorithm. We also question the reliability of the obtained output with respect to inconsistencies in Python interpreter versions and outdated libraries. Inspecting the intermediate format used as input for the disentanglement algorithm, we identified an inconsistency in the interaction with the pseudonymization script.

The version used to disentangle Discord messages by Subash *et al.* [244] is the same used by Chatterjee *et al.* for Slack [43]. To partially comply with user anonymization needs, user names of message authors in the ANNOT format have been randomly substituted with common first names. In this pseudonymization process, references to the authors in the text of messages have not been translated accordingly. This minor detail impacts the reply feature of the relatedness classifier, as reply links are completely lost. An ablation study could confirm this in the original algorithm but such a study is beyond the scope of this work.

7.6.2 Input JSON format

The JavaScript Object Notation (JSON) input format of CoDi (Figure 7.8) is richer than the *ANNOT* representation used as exchange format in the reference model [42, 69]. We provide an endpoint to convert between the two for compatibility and to cross-validate results from previous studies. The representation we propose better suits the features of modern IM platforms (*e.g.,* replies, quotes, mentions, attachments), and allows to more easily verify input consistency.

[5]See `https://tinyurl.com/mr3537ae`

```
"platform": "Discord",
"id": "b4138f14-af37",
"name": "Agile Everything",
"members": [ {
    "id": "d1ff9c5b-f1fc",
    "name": "Jermaine Fontaine"
  } , ...
],
"channels": [ {
  "id": "c65d238f-d987",
  "name": "visualization",
  "path": "agile/visualization",
  "topics": [ {
      "keywords": [ "Visualization" , ... ],
      "description": "Agile Visualization is cool!"
    }
  ],
  "messages": [ {
      "id": "34ce13f1-6577",
      "authorId": "d1ff9c5b-f1fc",
      "content": "Jermaine: $(date) is in large format",
      "conversation": "T35",
      "timestamp": "2022-02-06T19:24:23.777+00:00",
      "mentions" : [ { "authorId" : "d1ff9c5b-f1fc" } ],
      "repliesTo" : [ { "messageId" : "c1dr4c2r-z7at" } ],
      "attachments" :
        [ { "url" : "https://cdn.discordapp.com/..." } ]
    }, ...
```

Figure 7.8: JSON input format example.

7.6.3 Web User Interface

CoDi provides a web-based user interface (Figure 7.9).

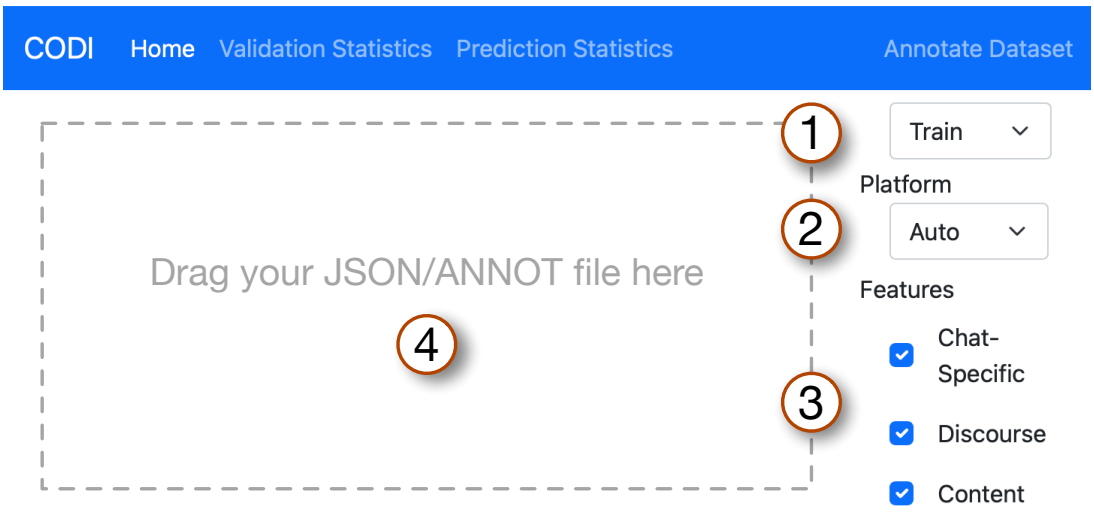


Figure 7.9: CoDi Home page.

In the *Home* page it is possible to select the operation to perform ① (*e.g.,* train, validate, predict), the type of platform ② (*e.g.,* IRC, Slack, Discord), parameters of the model ③ (*i.e.,* features to use in the classifier), and to drop a file for elaboration ④.

In the *Validation Statistics* page it is possible to compare the output of a disentanglement with a provided ground truth (Figure 7.10). Color coding helps in identifying conversation blocks. For example, the message from Khylon is misclassified and assigned to a new conversation (left) instead of being the last message of conversation 1 (ground truth, right).

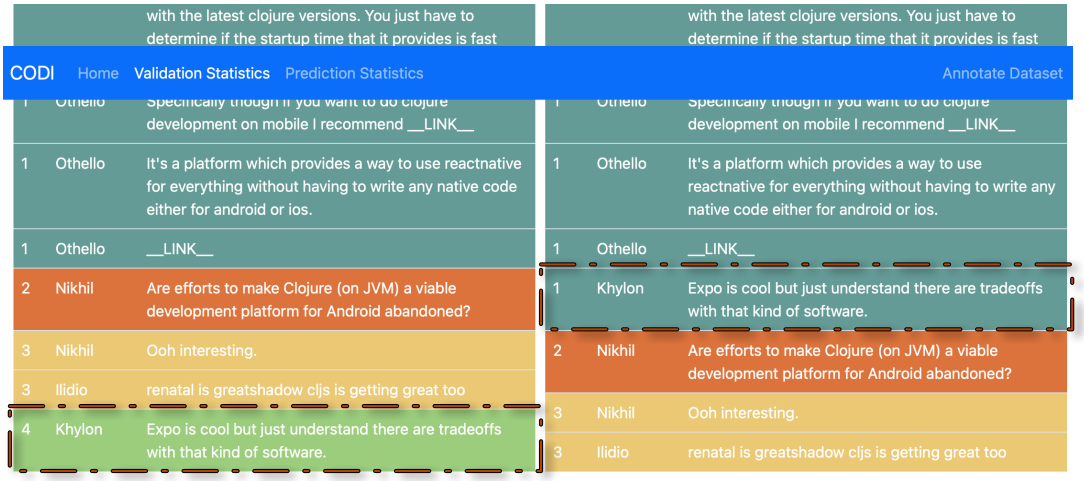


Figure 7.10: CoDi validation page.

In the *Prediction Statistics* page (already shown in Figure 7.6) it is possible to see the output of the disentanglement. Messages in chronological order (left) are mapped, with arrows, to their counterpart in disentangled conversation blocks (right).

Prediction and validation statistics pages have an expandable statistics section at the top (Figure 7.11). It provides detailed information on performance and other metrics for the output conversations as well as for internal components (*e.g.,* classifier accuracy, computation time).

7.7 Evaluation

To assess correctness of our implementation, we compared disentanglement annotations from CoDi (logistic regression relatedness classifier, all features) with the reference model [42]. We trained both models with the same ground truth: 3,544 manually disentangled messages from Chatterjee *et al.* [42]. We compared their performances on two previously published datasets [244] and a new one we extracted and partially manually annotated (see Section 7.7.1 and Table 7.4). We transformed datasets from the ANNOT format into the CoDi JSON format and vice-versa using CoDi's `api/convert` endpoint.

7.7.1 Datasets

Literature datasets: We used two subsets of messages from two Discord datasets published by Subash *et al.* [244]. The smaller one is the *clojure* Discord server dump (Feb–Apr 2020: 464 messages), the larger one is the *python* Discord server dump (Mar 2020: 56,763 messages).

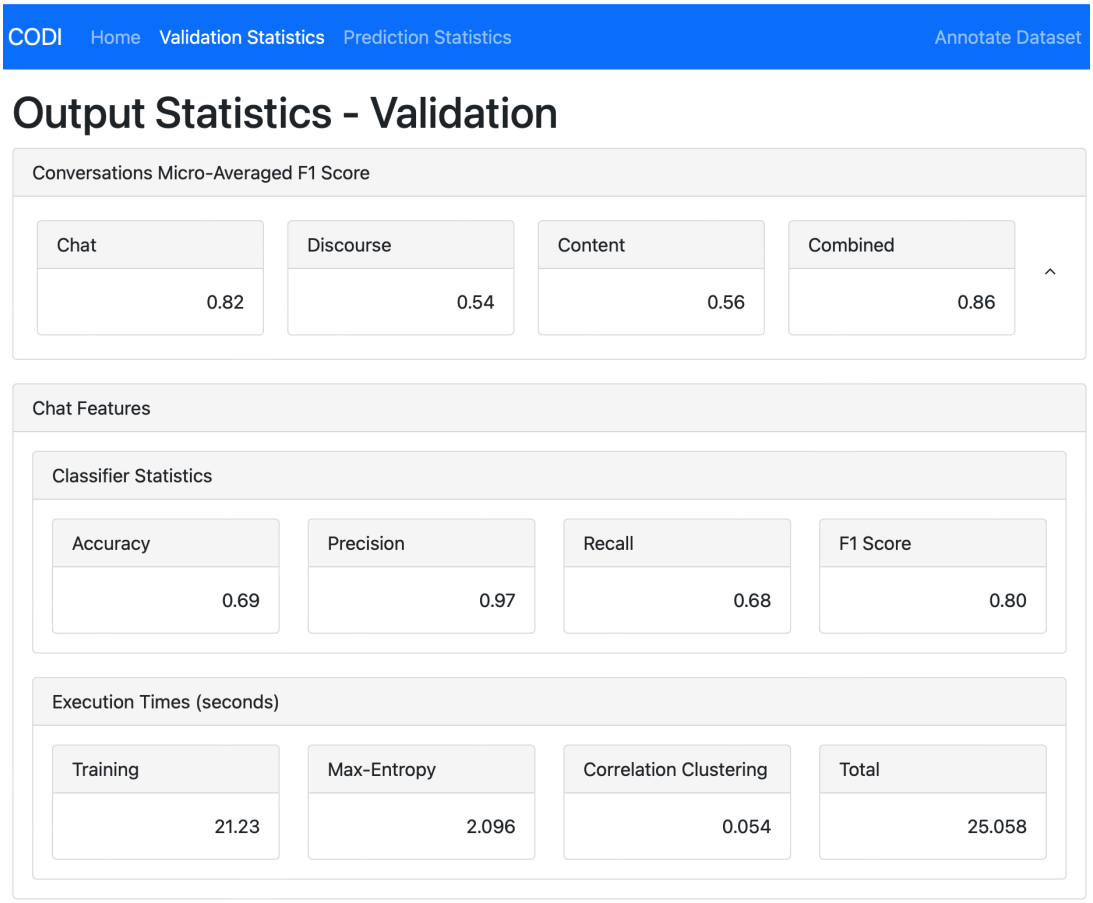


Figure 7.11: CoDi statistics expanded.

Roassal dataset: We extracted the full history of the *roassal* [6] channel in the *Pharo* [7] Discord server (Apr 2017–May 2022: 8,093 messages). To perform an independent testing on a new ground truth, two authors manually annotated four days of conversations in this dataset (*roassal_m*, 294 messages). The third author was involved in the discussion on conflicting assignments until consensus for all the messages was reached.

7.7.2 Result Analysis

In Table 7.4, we show the results of comparing CoDi's disentanglement with automatically disentangled conversations obtained from the reference algorithm (top-3 rows). We also report CoDi's testing performance on the new dataset we manually annotated as ground truth (*roassal_m* row).

We report accuracy, precision, recall, and F1-score of the relatedness classifier and the micro-averaged F1-score (μAvg-F1) [222] of the final clustering of disentangled conversations. We used the μAvg-F1 measure to evaluate correspondence between conversation clusters for comparability with previous results [42, 68, 69]. Moreover, a multi-way average over conversation clusters is suitable to compare automatic disentanglement to a ground truth and correlates with the one-to-one accuracy metric [69]. For python, due to the large number of messages in the dataset, we computed μAvg-F1 on a small random sample of 654 consecutive messages.

[6] See `tinyurl.com/roassal`
[7] See `https://pharo.org`

Table 7.4: CoDi vs. reference: Comparison and new test.

Dataset Comparison	Relatedness Accuracy	Precision	Recall	F1	Clustering μAvg-F1
clojure	67	94	68	79	**88**
python	60	86	62	72	**78**
roassal	77	95	78	86	**80**
New Test					
roassal_m	68	68	77	79	**63**

Correctness

High values of the μAvg-F1 score indicate disentangled conversations similar to those in the reference model. Our implementation suffers from lower recall of message relatedness. This indicates an overestimation of related utterances in the classifier. Lower μAvg-F1 scores for the python and roassal datasets indicate that there are still significant inconsistencies.

Problematic Cases

By manual inspection we found two main cases: *single point* and *split/merge* inconsistencies. In single point inconsistencies (Figure 7.12) one message is assigned to the wrong conversation. It can be the previous, the next, or a new one. Split/merge inconsistencies (Figure 7.13) happen when a conversation is split into two or two conversations are merged into one. Single point inconsistencies preserve most of the features of extracted conversations. Split and merge inconsistencies have a higher impact on μAvg-F1 but are often limited to a single conversation mismatch. Manual inspection confirmed the abundance of split inconsistencies with the python and roassal datasets. This is the main cause of the lower μAvg-F1 score with respect to clojure.

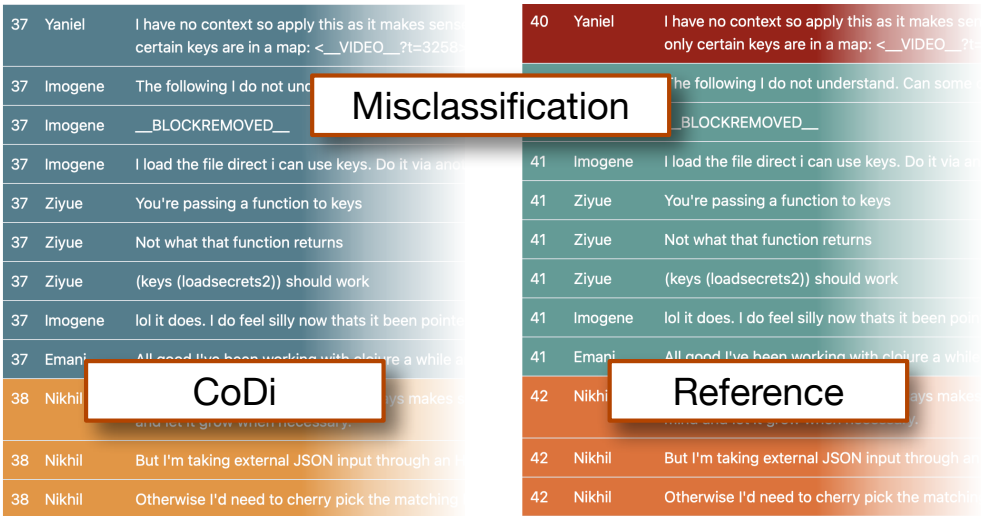


Figure 7.12: Example of single point inconsistency.

Generalizability

Testing with the roassal_m dataset shows lower performance in terms of μAvg-F1 score (reference implementation μAvg-F1 = 0.61) with respect to the results reported by Subash *et al.* [244].

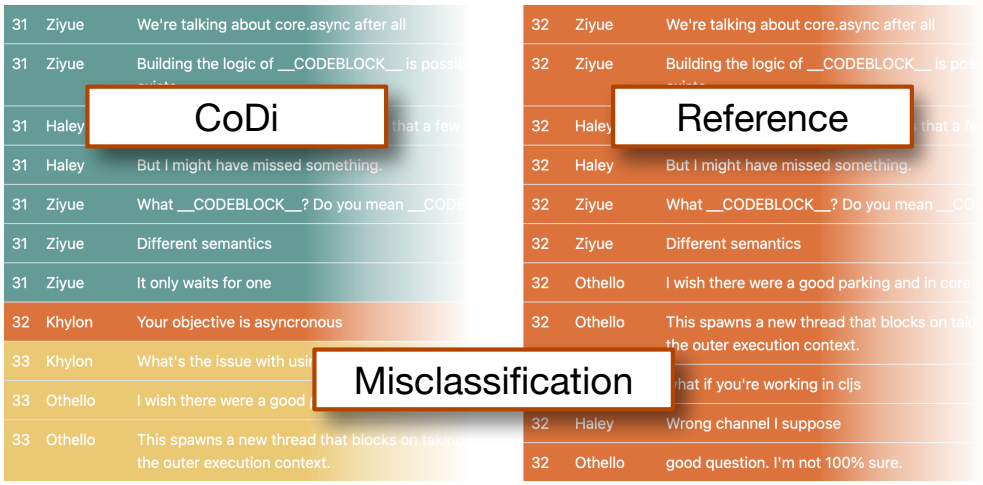


Figure 7.13: Example of split/merge inconsistency (split).

This might be caused by messages in the new dataset having different characteristics (*e.g.,* lower time dependence, different usage of mentions) and can potentially hint at low generalizability of the approach. Our study highlights the need to increase the amount of manually annotated disentanglement datasets to replicate previous studies.

7.7.3 Conversion and Formats

Chatterjee *et al.* used an Extensible Markup Language (XML) representation for the input [42]. The XML was converted to ANNOT format for compatibility with the original implementation. Due to the conversion, the XML format could not be exploited by the algorithm, thus remaining unexplored in terms of potential disentanglement improvements.

Format conversion also comes with limitations to cross-compatibility of input. For example, mentions in simple textual format are expensive for computing mention-related features. Providing them explicitly with a richer message representation, as already supported by CoDi's JSON input format, can improve performance and reliability of the disentanglement. In fact, the ANNOT format showed to be error prone in pseudonymized datasets due to inconsistent pre-processing steps (*i.e.,* name substitutions impairing mention traceability).

7.8 Future Work

We evaluated CoDi by comparing it with the reference model and on a dataset we manually annotated. We plan to extend the evaluation to the largest manually annotated dataset by Kummerfeld *et al.* [129] and explore improvements to platform-specific features (*e.g.,* quotes, replies). We are deploying CoDi for researchers to integrate it in their pipelines, which calls for an evaluation of the the user interface.

Configurability options, performance optimization, and robustness to malformed inputs need to be improved upon, with respect to the reference model and local approaches in general, especially on large datasets. CoDi can be deployed in pre-trained mode for multiple users. We will implement API-key access to support concurrent usage in segregated instances, allowing training custom models also on the publicly deployed service. Finally, we need to extend the model to another approach (*i.e.,* end-to-end) to demonstrate generalizability of our design choices for the JSON input format (Section 7.6.2) and extensibility of the architecture.

7.9 Summary

We reconstructed conversations in Discord as a first class concept, and presented an approach based on word-clouds and source code parsing to deal with their twofold nature. The approach could be adapted to other instant messaging platforms such as Slack and Gitter, *e.g.,* by supporting different APIs for extracting messages and content. There are also limitations in the available history of some communities (*e.g.,* Slack's free tier limit to the latest 10k messages). Different platforms could provide ways to retrieve more code blocks, even outside triple back-ticks or without explicit syntax highlighting. Nevertheless, we plan to explore regular expressions and machine learning techniques to pre-process messages and identify source code fragments that could be linked to conversations.

Conversation disentanglement is a key pre-processing step to improve knowledge extraction from instant messaging platforms. We need tools able to stop such fast and volatile sources from being only an on-demand support for program comprehension, and unleash their full potential.

We presented CoDi, an extensible object-oriented micro-service for conversation disentanglement, reporting on its comparative evaluation. Our work aims at improving the reliability of conversation disentanglement while reducing the technical barriers to reuse state-of-the-art models: CoDi is a first step towards making disentanglement models usable and accessible, complementing the needs of DiscOrDance to extract meaningful conversations.

8

Tool Support

In the previous chapters we already presented some of the tools we developed or co-supervised as an integral part of the research approach we propose to investigate the software documentation landscape. We leveraged Ragnadok, in Chapter 3, to gain preliminary insights on the documentation sources in GitHub projects. We presented DwarvenMail, in Section 4.3, and how it shed light on the tsunami of instant messaging applications currently reshaping the landscape.

A mix of data mining techniques, data and software visualization techniques, and a core domain model (actually, often object oriented) extended while gaining knowledge of the investigated subjects are also at the core of the two last studies that will conclude the presentation of our work on the documentation landscape. We focus more on the technical and visual aspects, but always driven by the insights that these tools elicit on the landscape and its evolution.

8.1 DiscOrDance

New communication platforms have emerged to support developers in finding and creating the knowledge they need for program comprehension, maintenance, and evolution. Instant messaging applications are supplanting developer mailing lists in collaborative development toolchains. These applications provide a new medium, supporting faster and richer communication (*e.g.,* embedded previews, images, files, videos). Research so far focused on extracting information from these platforms, but there is a lack of tools to visually and interactively explore them.

We present implementation details of DiscOrDance,[1] our tool for the interactive visual exploration of the complete message history of a Discord server. We show how three categories of views elicit insights on aspects of the structure, members, and software related content of a Discord server. We demonstrate use cases of DiscOrDance to support software maintenance and evolution activities on an active software developer community, the Pharo Discord server.

8.1.1 Visualizing Software Developer Communities on Discord

The Discord desktop application is organized in columns containing the servers list, categories and channels, the messages of the selected channel, and the member list. The mobile UI features the same components but with a layout more suitable for small screens. Discord servers may contain tens of thousands of users and reach a throughput of several messages per second (Figure 6.1).

DiscOrDance helps with high throughput and volatility of developer communications on Discord, enabling visualization and live exploration of the complete message history of server.

[1] Demo video: `https://youtu.be/eYCLGWwM9HY`

DiscOrDance's domain model is built around the source code and related conversations, to support software maintenance and evolution activities. The model is represented as a graph and it allows to extract and persist information in a form that better suits analysis and retrieval needs. Besides six pre-defined views, DiscOrDance can be easily extended with custom view specifications to elicit further insights. Metrics of nodes and edges (*i.e.,* entities) can be mapped to visual properties of the glyphs representing them (*e.g.,* rectangles, circles, lines).

8.1.2 Mining Discord

DiscOrDance is composed of a scraper bot, an object-oriented domain model of a Discord server, and a UI for interactive visualization and exploration of a server instance.

Figure 8.1 shows a screenshot of the user interface of DiscOrDance. The menu bar ① allows to select pre-defined view specs, change the layout of the current view, select nodes and edges and spawn a new view from the current selection, inspect the current graph, and handle visibility of specific elements. An entry also collects diverse custom operations related to the model (*e.g.,* cleaning, saving), metrics, and metric normalization. Entities ② are shown in the main canvas and can support on-hover tooltips ③ for displaying relevant information about them. DiscOrDance also supports contextual menu interactions on the entities (*e.g.,* right click on a message node → *Open in Discord*). A status bar ④ reports information about hovered entities and statistics for the graph in the current view ⑤. Finally, a selection window ⑥ allows to filter entities and select them programmatically.

DiscOrDance does not need any configuration and works out-of-the-box, but it can be extended by implementing new metrics and view specs (see Section 8.1.4 and Section 8.1.5).

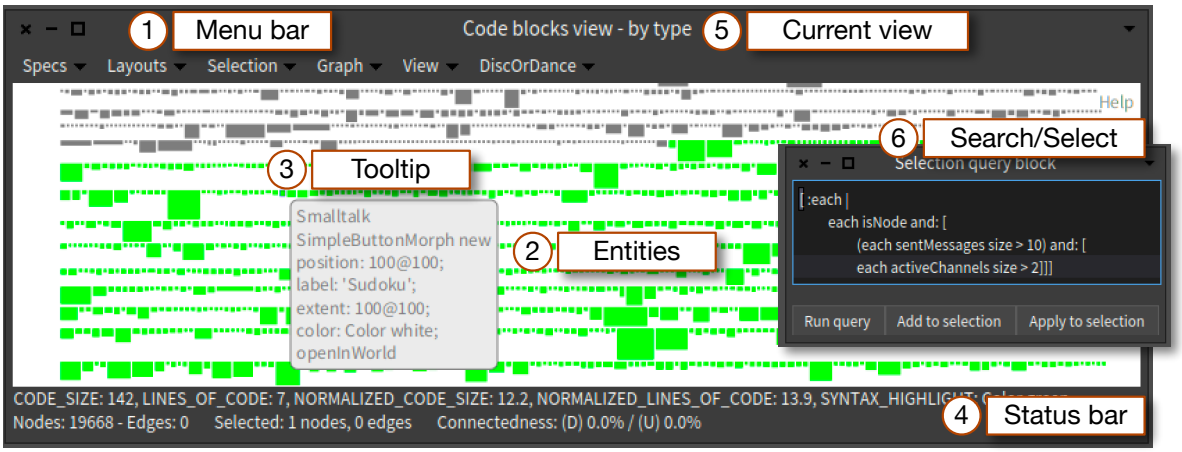


Figure 8.1: The user interface of DiscOrDance.

8.1.3 Architecture

Figure 8.2 summarizes the architecture of DiscOrDance.

DiscordST [41] is a client for Discord that includes a subset of the public Discord REST API. The bot must be added by the server administrators with reading permissions to be able to mine the full history of previous messages (*read_message_history*). Then it can scrape the structure and content of a server and store it locally. It retrieves channels and their messages, and builds the model (*e.g.,* adds links for mentions between authors).

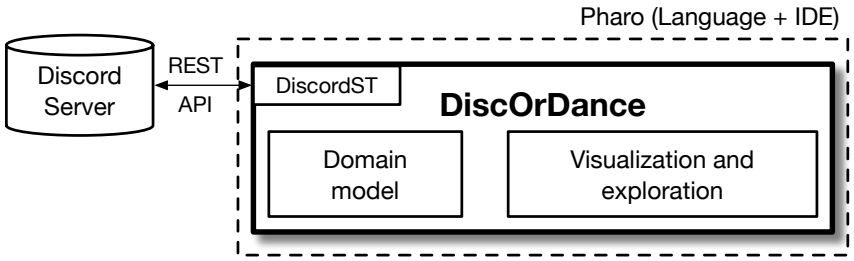


Figure 8.2: The architecture of DiscOrDance.

The scraping process attempts to retrieve all the entities of interest (*e.g.,* authors, messages, attachments, code) and populates the internal domain model accordingly (*i.e.,* nodes for messages, edges for mentions). After the scraping is complete, the graph of all nodes and edges is created and the resulting instance can be explored in a new window, according to a selected pre-defined view specification (Figure 8.1).

8.1.4 Metrics

DiscOrDance implements two types of metrics: Color and numeric. Color metrics map categorical attributes of the entity (*e.g.,* activity status) to an output color. Numeric metrics correspond to numeric attributes of the entity (*e.g.,* LOC of a code snippet). Table 8.1 shows the implemented metrics and an example of their meaning. Every numeric metric has a normalized version where the minimum and maximum values for the normalization can be provided.

Table 8.1: Metrics, types, and example values.

Metric	Type	Definition
Active Authors	Numeric	Number of authors active in a channel
Active Channels	Numeric	Number of channels in which author is active
Active Status	Color	Server membership status
Activity Span	Numeric	Number of days of activity in a channel
Channel Type	Color	Voice or Text
Class References	Numeric	Number of references of a class
Code Size	Numeric	Size of code in bytes
Contained Messages	Numeric	Number of messages contained in a category
LOC	Numeric	Number of lines of code
Max Daily Messages	Numeric	Maximum number of sent messages per day
Mentions	Numeric	Number of mentions of other users
Sent Messages	Numeric	Number of messages sent in a channel
Syntax Highlight	Color	Language syntax highlighting

8.1.5 View Specs

Views are the way entities of the graph are displayed in DiscOrDance. Views are created following templates called ***view specs***. A view spec is a specification of entities, a layout, glyph mappings, metric mappings, a sorting block, and a filtering block (all attributes are optional).

Entities: Limiting the scope to specific entity types allows to filter out all nodes and edges except those of interest.

Layout: Takes care of positioning entities on the canvas (unless a metric is mapped to the position of the glyph).

Glyph Mappings: Define associations between entity types and their visual representation. Custom glyphs are defined for nodes and edges. Subtype mapping is supported.

Metric Mappings: Each entity type can have associations between metrics of the entity (*e.g.,* number of references of a class) and visual attributes of the glyph (*e.g.,* size, color, position).

Sorting Block: A code block that provides the ordering of entities based on their properties. Layouts considering ordering (*e.g.,* flow layout) will position glyphs in the resulting order.

Filtering Block: A code block that filters specific entities based on their properties instead of their type. For example, a view considering only recent messages can show all message entities and filter according to sending date.

Although custom view spec definitions are possible (Figure 8.3), DiscOrDance offers six pre-defined views to explore different aspects of a Discord server: server structure (*Channel Activity View* and *Channel Activity Timeline View*), message authors (*Author Activity Status View* and *Author Activity Spark-line View*), and source code shared among members of the community (*Code Blocks View* and *Class References View*). These views are also implemented through the view specs mechanism and described in detail in our previous work [181].

```
self name: 'Class links view';
   layoutClass: RSRadialTreeLayout;
   entityClasses:
      { DDClassElement . DDClassNode . DDCodeReferenceEdge } asSet;
   glyphMappings: {
      DDClassElement -> DDSmallGlyph .
      DDClassNode -> DDNodeGlyph .
      DDCodeReferenceEdge -> VZEdgeGlyph }.
```

Figure 8.3: Example of a view spec definition.

8.1.6 Manual Inspection

The Pharo IDE provides an inspector for objects, supporting live navigation of object instances. Through the inspector we can explore the object-oriented model by navigating the properties and content of entities in the graph. The inspector also supports custom code execution in a playground to quickly try ideas and test hypotheses directly on the live model.

In Figure 8.4, we show an example of a code node glyph with its instance variables. The glyph is mapped to a code entity. Navigating the link to the container message allows us to see where that code has been referenced in a conversation and retrieve the actual message entity for further exploration. The Pharo code at the bottom can be evaluated in the context of the current object, for example allowing to open the original Discord message in the browser.

The IDE itself can be modified in Pharo, and the for DiscOrDance, we collect custom actions for the contextual menu of entities of the domain from the objects representing them.

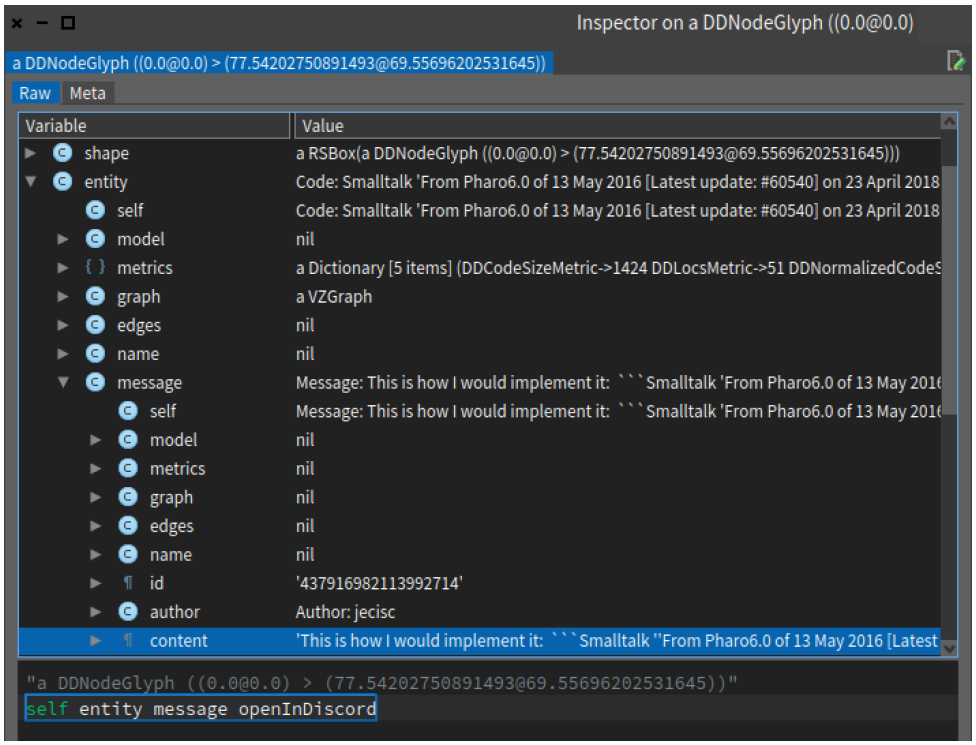


Figure 8.4: Pharo Inspector and an example code node glyph.

8.1.7 A Case Study—The Pharo Discord Server

We present insights from the Pharo community Discord server elicited by channel, author, and source code views. These categories cover the basic needs for insights on a Discord community.

Channel Views. This view provides a structural overview of a server. Categories and channels are depicted with shapes. Edges between channels and categories represent containment relationships. The size of glyphs is proportional to the number of messages, pinpointing the most active channels. Figure 8.5 shows the Channel Activity View of the *"Libraries"* category.

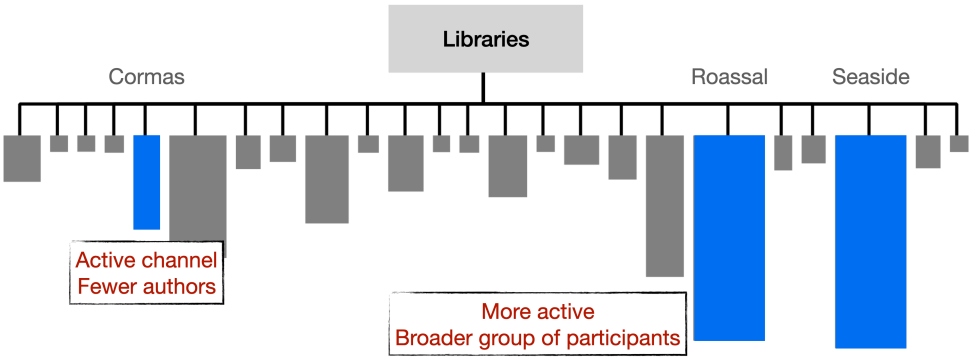


Figure 8.5: Channel Activity View of the "Libraries" category.

Its channels show different levels of activity and number of active members. By hovering over the *"Cormas"* channel, we can see in the status bar that it has 30 active authors who contributed about 3,100 messages. Channels *"Roassal"* and *"Seaside"* have a broader number of participants (*i.e.,* 154 and 153 respectively) with a higher activity of more than 7,500 messages each.

Figure 8.6 shows the Channel Activity Timeline View of fourteen channels, highlighting their different types. Either by manual inspection (Section 8.1.6) or with the information provided in the status bar (Figure 8.1 – ④), we can see that *"hadoop"* has been a short-lived channel while *"databases"* is old and still active (∼4,200 messages since Apr 2017). The Ukrainian Pharo community has been active between Apr 2017 and Dec 2019. French-speaking Pharo users received a dedicated channel in 2021: *"Pharo en français c'est ici* ☺*"*.

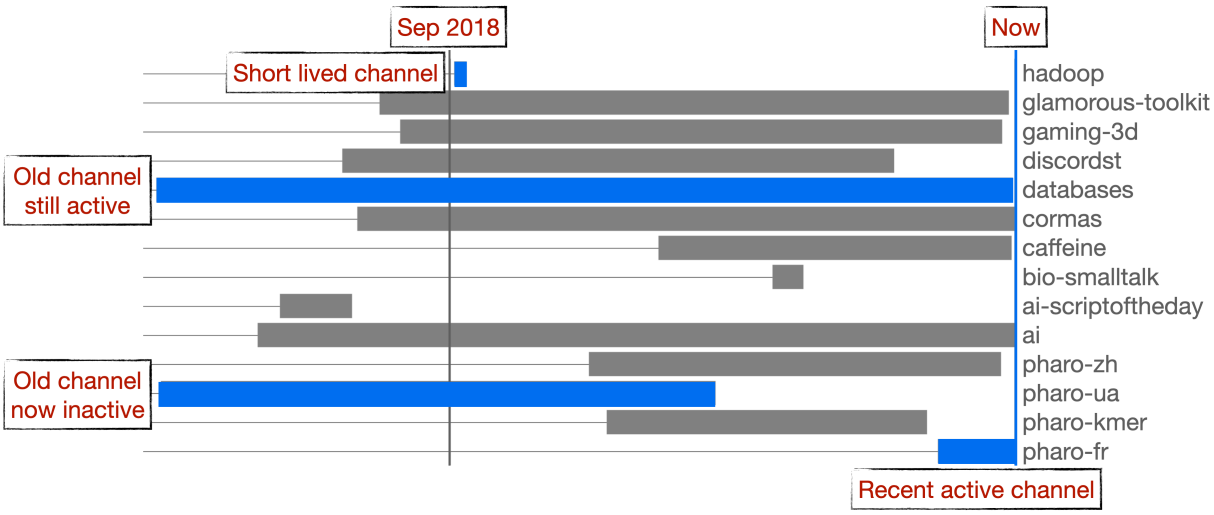


Figure 8.6: Channel Activity Timeline View of 14 channels.

Author Views. Author views map message authors to box-shaped glyphs. Edges connect authors based on mentioning tags (*i.e., @username*). Mapping the number of messages sent or the number of active channels to the glyph size allows for a visualization of the most (or least) active authors. An example view spec in this category is the *Author Activity Status View*.

By combining the selection query support with the Author Activity Status View we can quickly select 242 authors who posted a single message to the server, as depicted in Figure 8.7.

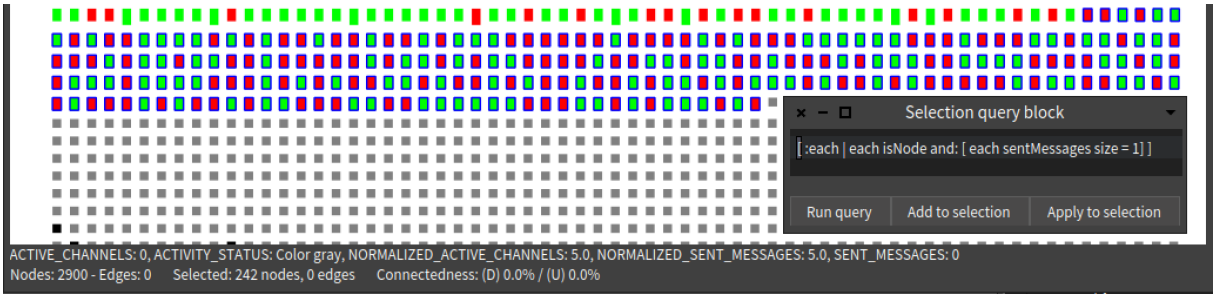


Figure 8.7: Selection query on Author Activity View.

Half of them are not part of the Pharo Discord community anymore (*i.e.,* they quit the server, highlighted in red). It remains to be investigated when they have been active and what is the nature of their only message. Answering these questions could help better address newcomers in the community. By selecting ex-members with a low number of messages, and manually inspecting their latest conversations, we could better understand why they abandoned the server. All these operations are already supported by DiscOrDance.

Source Code Views. *Code Blocks View* and *Class References View* are examples of source code views focusing on domain-specific content (*e.g.,* mentioned classes) to provide insights on language features discussed in the community.

In the Code Blocks View, such as the one depicted in Figure 8.8, the size of a code block allows distinguishing between short snippets and longer code. We can hover over different blocks to see their content. For example, one can find unformatted long lines of copy-pasted code by looking at the "wide" blocks, as these blocks have many characters in a few lines. In this view spec, the *syntax highlight* metric is mapped to the *color* attribute of code block entities. This allows visually estimating the ratio of source code snippets for each language. Length and language are only two examples, other source code metrics can be implemented and used in a Code Blocks View by associating them to the visual attributes of the glyphs in the view spec.

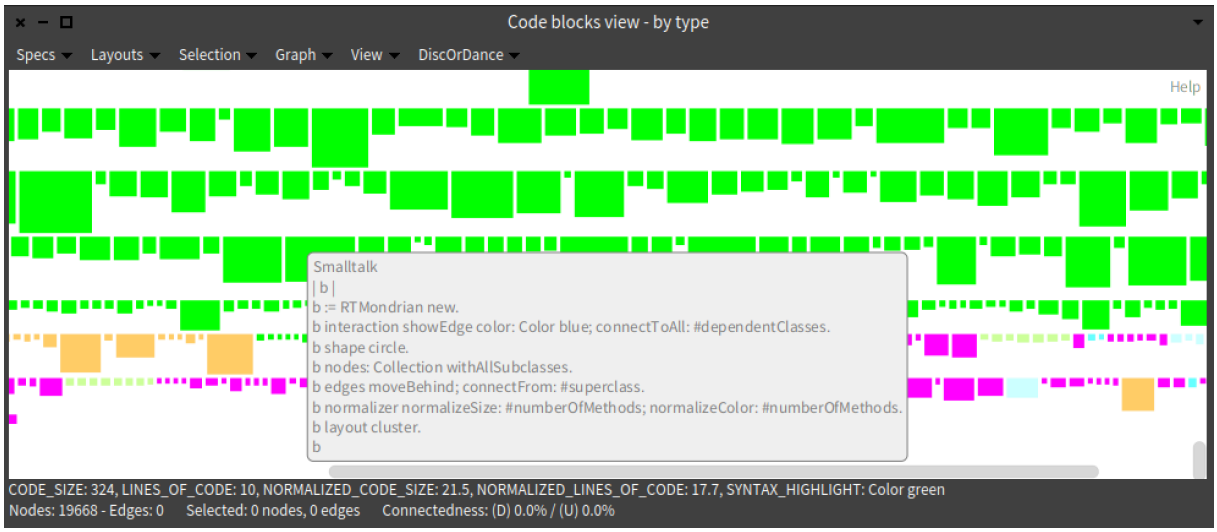


Figure 8.8: Code Blocks View showing code with tooltips.

Class References View is an example of a feature-specific view that has been implemented by extending the domain model. Code blocks are parsed to find class instances and new nodes are added to the graph for each class mention we find. Nodes are connected to the container message with code reference edges. The resulting view spec maps the number of mentions to the glyph size and uses simple labeled node glyphs. It can help us pinpoint the most discussed classes and find where their documentation could be improved.

8.1.8 Summary of DiscOrDance

DiscOrDance allows to explore the history of a Discord server. We described its user interface, architecture, and key implementation concepts. We defined view specs and metrics we used to provide insightful visualization of a server. Finally, we showed how we used it to investigate the Pharo Discord server and extract knowledge about its structure and contents.

DiscOrDance gives access to source code and conversations surrounding it. It is another step towards supporting developers in maintenance and evolution tasks by leveraging yet unexplored documentation in instant messages.

8.2 Drifter

We have already seen how UML artifacts constitute a key (but often neglected) asset supporting the comprehension of a system. Design documents "bind" developers in implementation phases and close the loop as documentation of the implemented system itself. Nevertheless, the intended system (design), its current version (implementation), and its documentation, naturally tend to drift apart, negatively impacting the usefulness of UML diagrams contained in such artifacts.

We present a novel approach to capture and understand the *Design – Implementation – Documentation (DID) drift*. We connect UML references in human-readable text-based UML formats (*e.g.,* PlantUML) to the corresponding source code entities (*e.g.,* Java classes), implementing novel metrics to capture the UML coverage of the system. We analyze project and file coverage evolution across releases and commits, with overall, method-level, and attribute-level detailedness, showing how they support DID drift analysis. We present interesting case studies exemplifying how through Drifter, the visual exploration tool we developed to validate our approach, we identify DID drift and ways to tackle it in the future.

8.2.1 Capturing and Understanding the Drift Between Design, Implementation, and Documentation

The Unified Modeling Language (UML) [81] is a visual modeling language that provides a standard way to describe (not only) software systems through a number of specialized diagrams. UML is used for design and documentation, providing a guideline for system development and evolution, thus constituting a fundamental support for the comprehension of the system's architecture and behavior, especially at higher levels of abstraction.

As software systems evolve and grow in size and complexity, they naturally diverge from their intended architecture. In the literature we can find multiple definitions circumscribing this phenomenon: Architecture erosion [16, 55, 137, 138, 139], architecture degradation [100], architecture consistency [8, 200], or architecture recovery [62] when attempting to mitigate the effect of an inevitable drift between the intended architecture and the implemented system.

These definitions miss an important aspect of the software development lifecycle. When a system is implemented, its current state must be documented and, although documentation is often an afterthought [6], design artifacts are the starting point for a high level comprehension of complex systems. In practice, what distinguishes a design UML diagram from one used to document the system, is time: Design-phase artifacts precede implementation, for example to describe a new feature, while a UML artifact conceptually becomes documentation only after it reflects the implemented architecture.

We investigate the relationships between design, implementation, and documentation. We model entities in two domains, UML diagrams and source code, to analyze how their artifacts drift apart from one another, creating a gap between the UML representation and the actual implementation (*e.g.,* classes covered by a diagram). We leverage the ease of parsing and the popularity of PlantUML, a text-based human-readable UML format, to associate Java entities (*e.g.,* classes, interfaces) with the UML diagrams mentioning them.

Structural relationships among entities and temporal relationships among activities on artifacts of the two domains (*e.g.,* file commits in the repository), are a form of drift "in space and time", providing insights on the nature, origin, and extent of the phenomenon we call *Design–Implementation–Documentation (DID) drift*.

8.2.2 DID Drift

We define Design–Implementation–Documentation (DID) drift as the distance between design and implementation and that between implementation and documentation. There are two types of DID drift: Space DID drift, stemming from the coverage metrics between UML and source code entities, and time DID drift, when we consider the temporal difference between modifications to artifacts that alter such coverage metrics, trying to capture how drift evolves.

We distinguish between overall coverage, and two detailedness metrics capturing finer-grained information: Attribute-level detailedness, and method-level detailedness.

Overall Coverage. The percentage of Java entities (classes, interfaces, enums) present in at least one UML diagram.

Attribute-Level Detailedness. The percentage of attributes in a Java class present in at least one UML diagram.

Method-Level Detailedness. The percentage of methods in a Java class present in at least one UML diagram.

For detailedness, if a method or attribute is covered in multiple diagrams, the percentage can be the minimum, maximum, or average of the respective detailedness for each UML diagram.

8.2.3 Dataset

Using the SEART GitHub Search (GHS) tool [52] we selected projects, excluding forks, with at least 2,000 commits, 10 contributors, 100 stars, and 10k *cloc* [333] LOCs to remove toy projects. We had a starting dataset of 13,152 repositories, which we cloned locally.

UML File Tagging. To tag UML files in our dataset, we follow the same approach described in Section 5.4. File extensions are an efficient way to discriminate file types, but not sufficient to uniquely identify UML diagrams. Thus, to identify UML artifacts, we combined: Import/export extensions from tools returned by searching for "*top UML diagramming tools*" on Google, all extensions with "`uml`" in the name and all extensions in any file path with "`uml`" in the name, when present in at least two repositories. We removed any known non-UML extension (*e.g.,* .java, .jar, .am). We implemented strategies to manually find examples and counter-examples of UML diagrams for each extension. For each potential UML file extension, we devised heuristics based on regular expressions to tag the files that contain UML semi-automatically. After identifying UML extensions and tagging UML files, we obtain the final dataset of 552 repositories containing UML diagrams. Table 8.2 summarizes descriptive statistics of the projects we present as examples and case studies.

Table 8.2: Statistics of the case studies.

Project	Commits	Contributors	Latest Release		
			Parsed Files	Java Entities	UML Diagrams
orekit	8,594	52	1,344	1,215	55
teammates	18,369	609	475	450	15
dataverse	27,492	179	811	806	13

8.2.4 DID Drift Analysis

The *git* log contains information about Java and PlantUML files for each commit.[2] After extracting from the log information about all .java, .puml, and .plantuml file modifications, we employ two parsers to analyze their content in all the versions. We parse Java files with the *javalang* library.[3] To optimize the parsing we adopt a differential parsing strategy by considering only the differences between two commits. We parse PlantUML files with the *plantuml-parser*[4] to extract references to Java entities from UML diagrams.

We end up with two instances of similar models, one for Java and one for UML, both having *packages, classes, interfaces, enums, references, methods, fields,* and *arguments*. To accommodate for UML's flexibility and get closer to the representation of Java systems, we need to disambiguate parameter types in UML method signatures, based on type separators (colon) and capitalization. We connect references in UML to entities in source code in an undirected graph (see Section 8.2.4). We extract information about releases[5] from GitHub, through its REST API, and add it to the cloned projects summaries.

The result is Drifter, an interactive explorer to capture DID drift in GitHub Java projects. Drifter allows to analyze a project and select a GitHub release or a *git* commit. It presents four data visualizations, each pertaining to a different aspect of DID drift: Coverage and detailedness, relationships between UML and source code, and coverage evolution at project and file level.

Package Visualization

Figure 8.9 shows UML overall coverage and method-level detailedness (Section 8.2.2) of the *csi/orekit* GitHub project. Each innermost circle represents a Java entity (*e.g.,* class). Entities covered (*i.e.,* mentioned) by at least one UML diagram are green, those with only an implementation are white. Java entities are contained in outer circles corresponding to packages and their nested containment relationships.

Method-level detailedness coverage goes more in depth by considering the percentage of methods implemented in a class that is covered by the corresponding UML reference. Since multiple diagrams can contain partial references, we provide different aggregation types for detailedness metrics (*e.g.,* min, max, average).

Example Insights. While we can find some packages with a good overall coverage (*i.e.,* most classes are green), a low method-level detailedness of the same classes reveals that the use of UML is partial and oriented to an overview of the system architecture. When classes in some packages have 0% method-level detailedness but most of them are shown as covered in the overall coverage visualization, it likely indicates the presence of a package diagram, only mentioning the involved classes without detailing them.

UML–Source Graph

Figure 8.10 shows Drifter's UML–Source graph, where references in UML diagrams (green) are connected to the corresponding Java entities (blue). Graph connectivity visually represents how many diagrams a Java entity is referenced by, and how many references to Java entities are contained in each UML diagram. Undocumented Java entities are represented by unconnected nodes in the UML–source graph.

[2]Linearized history, commits in topological order (`git log --topo-order`)
[3]See `https://github.com/c2nes/javalang`
[4]See `https://github.com/Enteee/plantuml-parser`
[5]Releases correspond to *git* tags but are mined from the GitHub API separately.

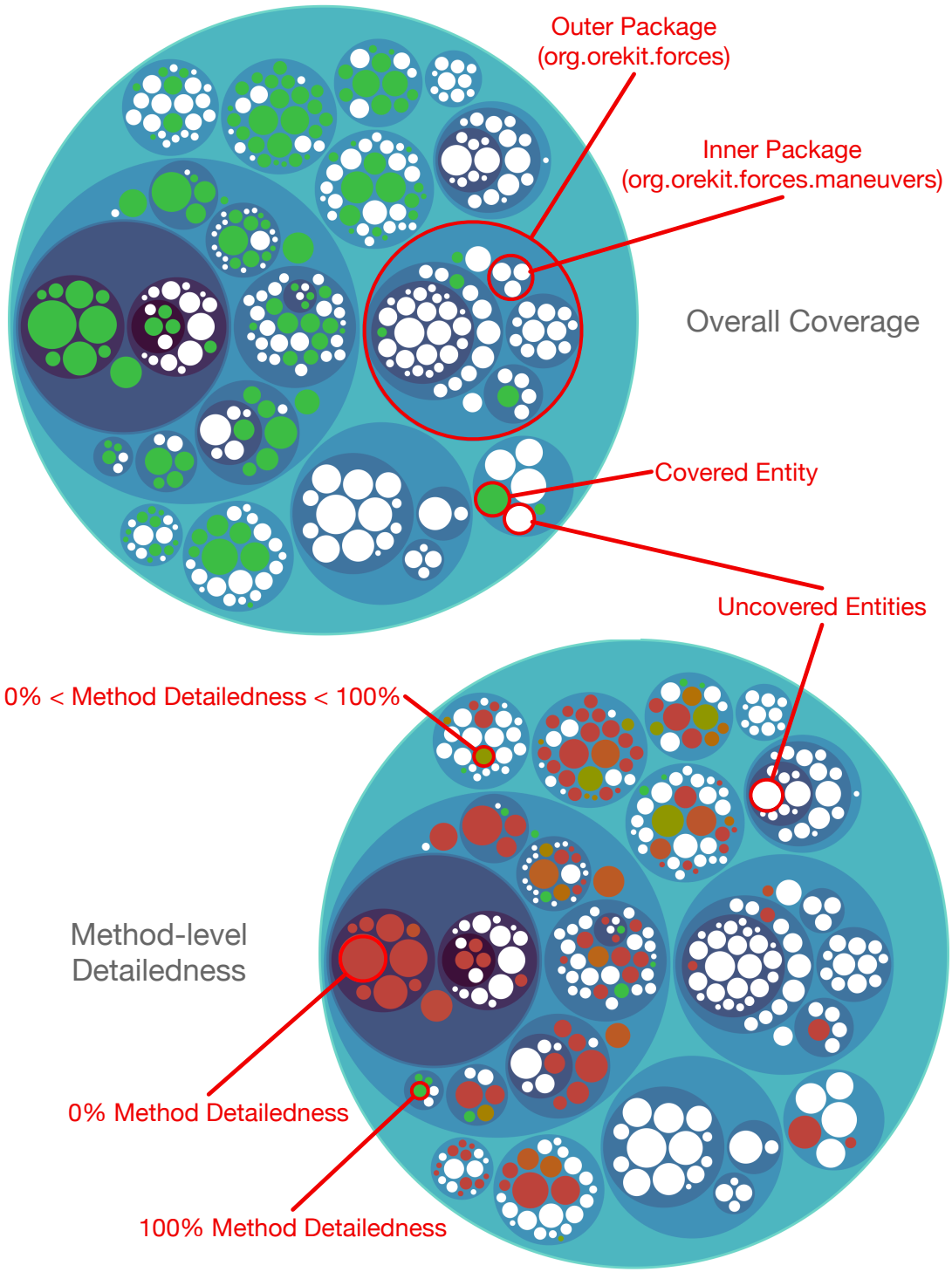


Figure 8.9: Package visualization.

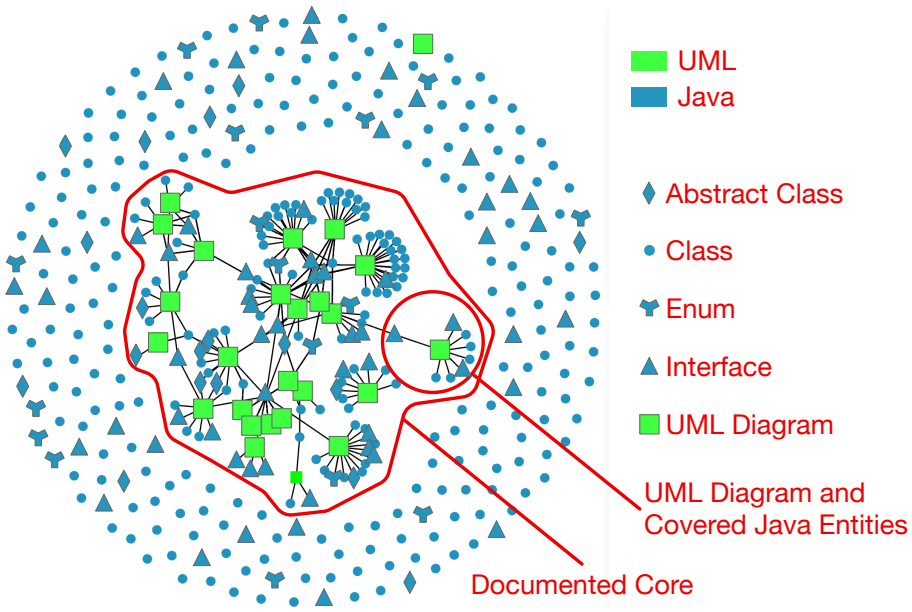


Figure 8.10: UML–Source graph.

Coverage History

The package and UML–Source graph visualizations are useful for exploring a specific commit of a repository, but they lack support for understanding the evolution of UML coverage, especially at a higher level. Figure 8.11 shows coverage history percentage over time with respect to the total number of classes in the system.

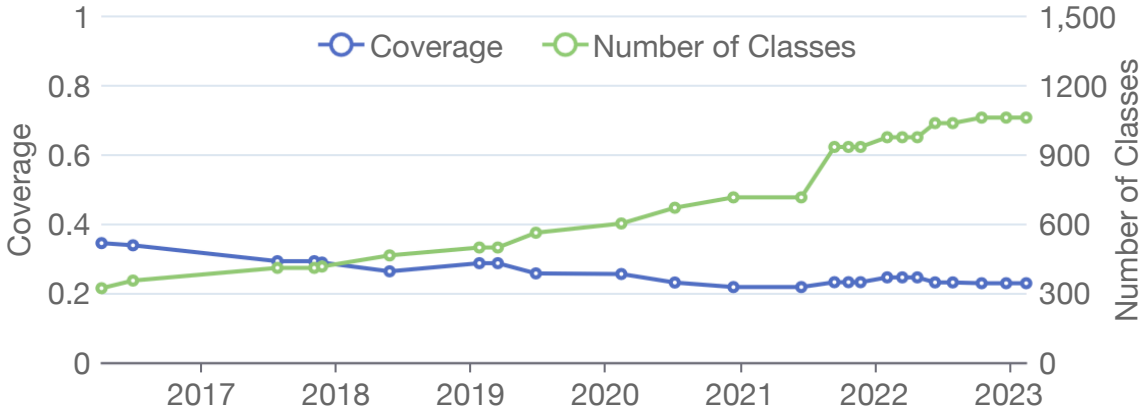


Figure 8.11: Coverage history chart — Release view.

Release view supports a coarse-grained representation which abstracts from single commits and low level details of the development workflow. Commit-level coverage history is more fine-grained and can be leveraged to assess the current status and guide the development workflow and resource allocation. If, for example, the coverage drops significantly, to avoid increasing undocumented classes, existing diagrams should be updated or new ones should be created. Release view tends to be more consistent thanks to the fact that, when there is a release, the project should be in a good, stable, buildable, and therefore more reliably parsable state.

File History

Another relevant evolutionary aspect is how a single file changes over commits and releases. We compare the evolution of Java source code files with that of the UML diagrams containing a reference to the same entities (*i.e.,* the corresponding Java class).

Figure 8.12 shows a zoomed-in view of the file history for *Instructor.java* in the *teammates/teammates* repository. The symbols mark the commit index where each file has been added (squares), modified (circles), or removed (diamonds). A red line for Java files indicates a lack of coverage for the Java entity in the corresponding commits.

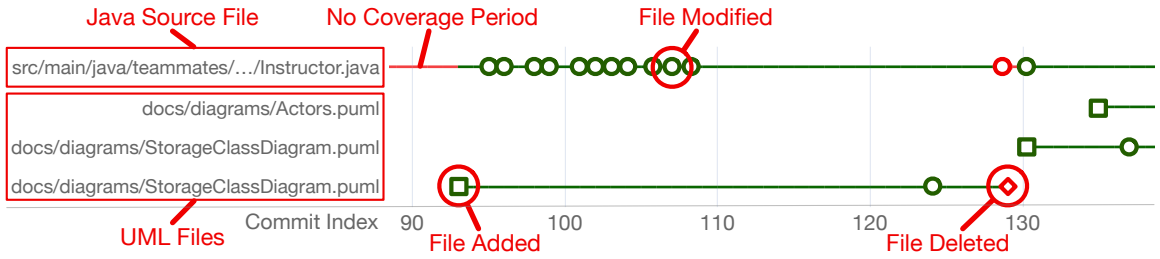


Figure 8.12: File history for *Instructor.java*.

Example Insights. The Java source file history has a period of no coverage (red line, top-left). The *Instructor* class was not mentioned in any UML diagram until it appeared in the new file *StorageClassDiagram.puml* in commit 94 (April 2021[6]). The file was removed and added again in two subsequent commits, temporarily leaving the Instructor class uncovered. We can also notice how 11 modifications to the Instructor class after commit 96 did not have a corresponding modification to the StorageClassDiagram.puml file, indicating that either only implementation details were changed or the UML documentation could have become outdated. Finally, in commit 135, *Actors.puml* mentions the class which is therefore referenced in two different UML diagrams.

8.2.5 Case Studies

We present in two case studies how we capture and understand DID drift from the interaction of multiple aspects in our characterization of the phenomenon, how our time-based approach allows identifying systems where design precedes implementation, but also contrasting wishes and reality of UML artifacts' maintenance.

Teammates

Teammates is a web-based peer feedback management system for students and teachers used by more than 800,000 users from over 1,110 universities around the world.[7] We used the Teammates project to present example insights elicited by the file history analysis (Section 8.2.4). Now we focus on the coverage history of Teammates in the last 2.5 years (Figure 8.13).

With the release 7.15, the project's documentation was migrated to PlantUML files. The chart shows an increase in coverage to 6% (22 classes out of 368) in a minor release and does not increase thereafter. In fact it reaches 5% when the system comprises up to 446 classes. In version 8.25.0, the same 22 classes are covered by 4 PlantUML diagrams. The UML–Java class graph confirms these findings.

[6] Drifter supports tooltip-based commit inspection (*e.g.,* date, committer, message).
[7] See `https://teammatesv4.appspot.com/web/front/home`

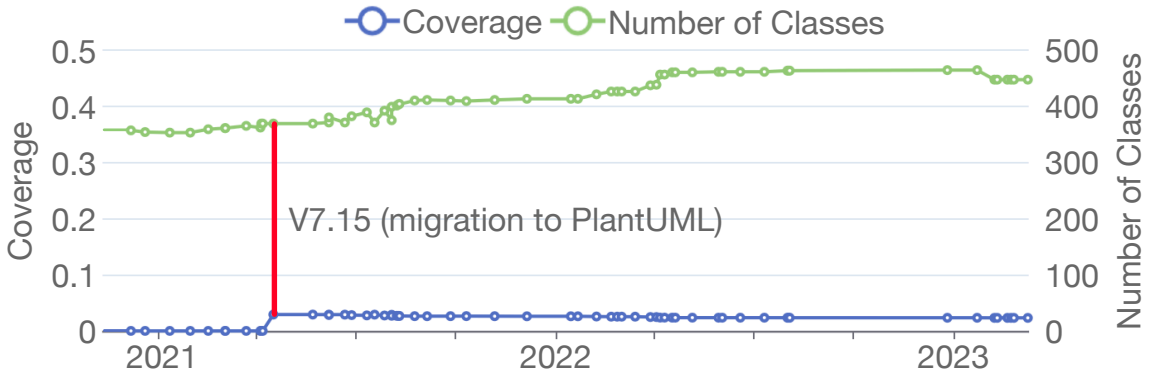


Figure 8.13: Teammates release coverage history.

In the last two years, not a single class has been documented in a .puml file, despite the decision to, paraphrasing from the discussion in the migration issue, adopt PlantUML for being free and usable without the need of specialized software, version control friendly, and better supporting a more consistent use of UML notation.

Similar motivations are found in another case study, *cs-si/orekit*, and are a possible reason why PlantUML is becoming popular. Nevertheless, these motivations did not correspond to an increase in the documentation effort. We also performed an analysis of the maintenance and maintainers of existing UML diagrams but this is outside the scope of the presented work.

Dataverse

Since the late adoption of PlantUML in the Teammates case study limits our analysis of the DID drift, especially in the early phases of the project where we assume a higher focus on design, we found another project that started with .puml files as its form of system documentation. Dataverse is a software platform for sharing, finding, citing, and preserving research data, and it is managed by the Institute for Quantitative Social Science (IQSS) at Harvard University. The *iqss/dataverse* project is the perfect example of "design before coding" approach.

The commit-level coverage history highlights 3 "coverage events" (spikes in coverage percentage, detailed in Figure 8.14). We perform an analysis of the Java files and UML Java references added in the commits constituting two of these events.

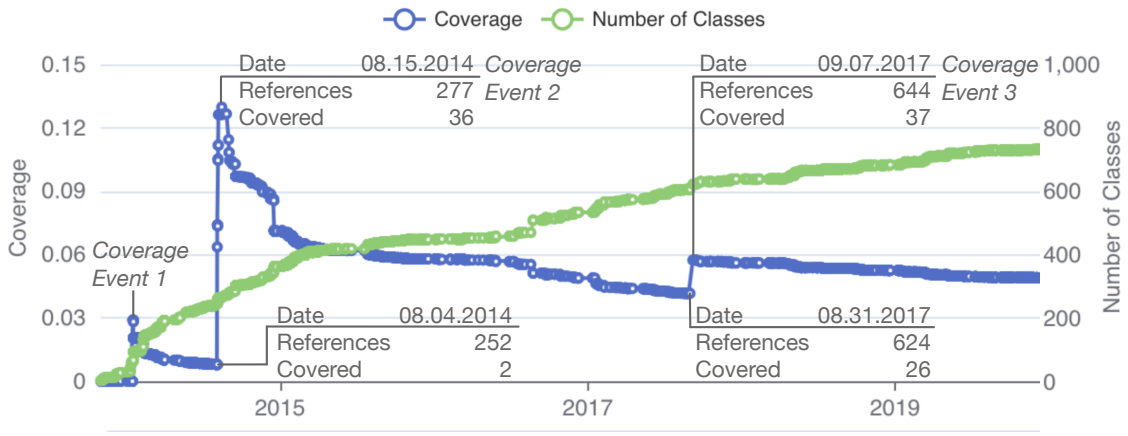


Figure 8.14: Dataverse commit-level coverage history.

Coverage event 2 sees the addition of 25 new Java classes that increase the overall coverage, indicating that those classes were already present in some UML diagrams as a design of the system that has been subsequently implemented. Nonetheless, there are 9 additions to UML diagrams contributing to the coverage increase by documenting previously undocumented classes. Coverage event 3 is more balanced in terms of design versus documentation, with 5 newly covered classes in UML diagrams and 6 new class implementations already covered by design. The Workflow class is documented and implementations of its collaborating classes WorkflowContext, WorkflowStep, WorkflowStepData, WorkflowStepResult, and WorkflowStepSPI are added to the source code.

Discussion

The case studies highlight the relevance of evolutionary analysis to assess DID drift. Manual analysis is supported by the proposed approach in highlighting points of interest in the development history. We also provide an interpretative framework to formulate hypotheses on the three-way interactions between design, implementation, and documentation leading to drift.

The Workflow- classes in the Dataverse case study indicate a discussion about the design, persisted in a UML artifact, before starting the actual implementation. The procedure to identify these "*design-first*" approaches could be automated to perform a large scale study on UML practices, to evaluate its usefulness and usage preferences as design or documentation tool.

8.2.6 Summary of Drifter

Coverage and detailedness provide information about UML as documentation but also when UML is used for design. The temporal relationships between UML and source code entities indicate whether UML is used for design or documentation. The presented project-level evolutionary analysis hints at how to make DID drift actionable to improve the quality and usefulness of UML artifacts, in turn improving system comprehension.

More needs to be done to exploit the potential of DID drift analysis. For example, evolutionary analysis to distinguish between UML for design versus UML for documentation needs to be made automatic. Support for languages other than Java and PlantUML can also improve the applicability of our approach. This work constitutes a necessary first step towards a better understanding of the interactions between UML for design and documentation, its drift from implementation, and how to mitigate it efficiently.

8.3 Summary

We presented different tools that we developed or co-supervised and that allow going in depth in one or more of the documentation sources of the landscape. Each tool tackles part of the landscape at a different scale, focusing on a specific source instance, like Discord, an entire category (*e.g.,* UML documentation artifacts), or even a documentation archetype, like in the case of Dwarvenmail's analysis of the Community archetype and its evolution. These scales are important to highlight phenomena of different amplitude and which entail different risks and opportunities for software practitioners and the software engineering community.

The tools are just a means to investigate the landscape at different levels of detail, the insights elicited by these tools through our analyses are the key to improve software documentation practices. DiscOrDance scratches the surface of what could be done with proper tools, for example, to identify trending features of a programming language discussed by its community, even before they can be reliably tracked in source code committed in software repositories.

The tools developed to reify and investigate the software documentation landscape highlight in a very concrete way the challenges that we need to overcome to successfully leverage the information contained in documentation sources. Just as an example, the level of granularity of the modeled information in DiscOrDance plays a crucial role in the information extraction and its further processing. With progressive refinements (*e.g.,* from messages to conversations, from time related conversations to disentangled ones) we complement the breadth oriented overview of the landscape presented in the second part of this thesis.

Finally, Drifter demonstrates how we should keep in mind the final goal of reconnecting documentation to the ground truth about it: Source code. By analyzing different artifacts, pertaining to different stages of the software development lifecycle, we can follow their evolution. Drifter focuses on discovering and understanding problematic evolution patterns, the next natural step is to take action to prevent or correct them.

Part IV

Epilogue

The End and the Beginning

9

Conclusion and Future Work

The software documentation landscape is a heterogeneous multitude of documentation sources, constantly evolving, shaped by forces beyond the control of the single developer. With our work, we highlighted some of the promises and perils that the paradigm shift currently affecting the landscape entails. We discussed also the limitations of the approaches we used, like the limited discoverability of secondary sources, and the computational complexity of reconstructing arbitrarily complex histories of files that are not identifiable as landscape sources upfront.

The software engineering community, so far, focused on the single sources, on comparing a few of them, and on porting and adapting approaches that work for one source to others. We advocate for the adoption of a holistic interpretation of the documentation landscape as a complex and evolving mix of intertwined single sources.

9.1 Past Contributions

We provided a consistent mapping of the landscape across multiple dimensions. We characterized its sources in terms of form and content. We went a step further for the categories of interest that are becoming the de-facto standard for software documentation, *at least* in open source software. We investigated Discord as an instant messaging application that recently has become a discussion hub for, literally, thousands of relevant GitHub projects and hundreds of thousands of developers and practitioners.

The evolution of the UML landscape and the resurgence of UML as a documentation means, under the pressure of *new* human readable textual formats, brings *new* possibilities for research. One above the rest is the ease of parsability of these formats with respect to their graphical counterpart, which played a fundamental role in our investigation and in the creation of new tools for the exploration of this non-negligible part of the landscape. Moreover, some guidelines emerge from looking at the bigger picture, using the landscape as an evolving overview over documentation practices also provides valuable insights on how to influence this evolution.

Looking at our work in terms of discovery and classification of documentation sources (and in particular at what we presented in Section 5.1), the analysis of the *instances* points to clear directions, although with less than clear paths, for the future of tools and specifications to create and maintain UML artifacts, like, for example, PlantUML and Mermaid.

From a technical point of view, the numerous tools we developed, co-developed, and whose development we co-supervised contribute to a software ecosystem to investigate the documentation landscape, on top of which other researchers can build.

We made some of these tools publicly available, along with the relevant datasets, as open source software, like DwarvenMail[1] and DiscOrDance[2] (with its graph visualizer component Vizor[3]). We deployed others to allow researchers experimenting with them as *configuration-less services*, like Drifter[4] and RagnaDok.[5] In some cases we did both (*e.g.,* CoDi[6,7]).

The heterogeneous nature of the documentation landscape is also reflected by the heterogeneous nature of the research lines presented in this thesis. An advantage of this eclecticism is the abundance of ideas to explore in each different direction.

9.2 Short-Term Future Work

Every tool we developed has many potential improvements, usually tied to technical aspects, that could increase reliability, generalizability, or usability. Apart from those technical enhancements, we would like to focus on new contributions that are just "one step away".

9.2.1 Replication and Extension of the Preliminary Study

Although we presented multiple snapshot views of the evolution of the documentation landscape, we aim to be able to animate it and convey the changes in a more imminent way. We will improve the fine-grained information (see the computational limitations for the time bucketing when analyzing changes in the aggregate landscape in Section 3.4.2) and the interpretability of shifts at a higher level of granularity (macro-scale phenomena).

Revisiting and extending the work presented in Chapter 3 will allow to further refine the taxonomy, possibly adding new recent sources. We plan to investigate more in detail the evolution of the last few years, focusing on a fine grained analysis as we did for the IM category,to capture the changes and trends in the *current* software documentation landscape at large. Finding evidence of emerging sources, absent in the taxonomy two years ago, could further validate the landscape as a means to understand new phenomena worth investigating.

9.2.2 Trending Sources and the Scientific Literature

The current representation of the landscape allows comparing the rise of popularity of some sources with their investigation in the scientific literature. Finding evidence of a temporal relationship (*i.e.,* the rise in popularity of a source precedes the scientific literature or vice-versa) will clarify if the software documentation landscape can be used to predict trends at source level or if it is useful only to represent and explain larger shifts affecting multiple sources at once.

We demonstrated this in the Discord case studies. The advent of Discord presented in Section 4.7 shows how, despite the very few scientific contributions to the investigation of this documentation source, it has become way more popular than the more studied counterparts (*e.g.,* Slack, Gitter). We argue that this trend is the norm and that software documentation research is playing "catch-up" with the practices in the industry. The software documentation landscape as a whole (and the tooling we developed) could help tip the balance back towards adoption of research-informed documentation practices, built along successful tools and not after them.

[1] See `https://figshare.com/s/33c8af534dba61d72c41`
[2] See `https://github.com/USIREVEAL/DiscOrDance`
[3] See `https://github.com/USIREVEAL/Vizor`
[4] See `https://drifter.si.usi.ch`
[5] See `https://ragnadok.si.usi.ch/`
[6] See `https://github.com/USIREVEAL/CODI`
[7] See `https://codi.si.usi.ch`

9.2.3 Invisible Documentation

The systematic mining of GitHub readmes to find documentation sources linked by project developers (Chapter 4) provides a view of the *surface* documentation. A part of the landscape remains "invisible" (to our mapping) for two possible reasons.

Either the source is so imminent for the repository that it needs no mentioning to be found, or the project uses its website to link other documentation sources instead of including them in the repository itself. The former case is exemplified by GitHub Issues in GitHub repositories. Not all readmes of projects using GitHub Issues document them explicitly, even if they use them. By identifying such sources from at least one repository, we can include them in the landscape and develop strategies to look for their content anyways.

The latter case needs more effort. We plan to investigate the coverage of the landscape by what is mentioned and what is actually present. One of the main goals is to provide information on how to improve the discoverability of the project's documentation through the landscape. Finally, we will scrape and analyze some of the primary sources (*e.g.,* website of the project) to look for secondary ones and systematically map them with approaches similar to the ones we described throughout this thesis. This will further validate the robustness to input changes of our methods (*i.e.,* using REs to capture links in the project's website for known sources).

9.3 Long-Term Vision

We discuss the two most salient points that emerged from our analyses as critical for the evolution of the concept of the documentation landscape, way beyond just an explanatory metaphor. Software documentation is an asset for a software project. Curating the landscape means to give an improved discoverability to important documentation, thus increasing the value of this asset.

9.3.1 The DoLMaR Specification

One of the main limitations of our mapping approach is the sensitivity of the REs we employed for tracking the links pointing to documentation sources. To mitigate this limitation, we employed the most restrictive RE yet capturing what we found as patterns for the source. We complemented this with a priority system for the REs to be as selective as possible in identifying sources in the different levels of the documentation landscape's taxonomy.

We still believe that a fully automatic approach to the mapping of the landscape would be preferable to reduce errors and outdatedness of this meta-documentation (thus suffering from the limitations of documentation). Nevertheless, we ponder the advantages of an explicit representation of the landscape of each software project as an unambiguous machine readable specification. We refer to it as the Documentation Landscape Machine Readable (DoLMaR) specification. Discoverability and reliability of the landscape are the two most prominent aspects that would benefit from the DoLMaR specification. We argued in favor of the usefulness of the landscape, for the single project and as a whole. We showed examples of how they can be leveraged to understand micro and macro-phenomena affecting modern software documentation.

In the future, we will focus on defining the DoLMaR specification. We can leverage the insights of our previous works to create recommender systems to partially automate the creation of the specification artifact for a project and to simplify its maintenance.

9.3.2 Dimensions of the Landscape

More work is needed to explore the interplay between multiple dimensions of the landscape. We showed how there are orthogonal dimensions that characterize a source. What is yet to be fully understood, is how the current and past sources are scattered across this n-dimensional space. Clustering sources by category in the taxonomy helped in defining one possible grouping. There, instances emerged naturally from our open card sorting approach to indicate the polarization of developers towards certain platforms for specific documentation needs.

This polarization is an implicitly known (and exploited) phenomenon in the SE community. An example of polarization is the rise in popularity of StackOverflow as a crowd-sourced and crowd-curated documentation repository that spawned many works investigating the potential of this source to provide reliable software documentation.

What remains to be addressed is how to anticipate or systematically investigate the new trends, as soon as they emerge. The research efforts should not try to catch up with technological advances and de-facto standard adoptions. We should anticipate and influence them towards profitable but sustainable evolution paths.

In our example of the evolution of Gitter, Slack, and Discord (see Section 4.7, and in particular Figure 4.8), the highlighted repeating trends could inform about the new platforms that are already becoming relevant for multiple software project. This can happen only if we look at the landscape as a whole and compare the relative growth of some sources in the recent past, potentially looking for specific patterns (*e.g.,* slow adoption and sudden rise of popularity as for the three aforementioned platforms). Moreover, as our example on the rise of UML text-based human readable formats highlighted, by identifying relevant sources, we can identify problems affecting them. By comparing with the other sources in the landscape, those being abandoned in favor of the new ones, we can derive features, along some of the dimensions of the software documentation landscape, that play a role in these popularity shifts.

9.4 Uncharted Landscape: LLMs for Software Documentation

When I started my PhD and for almost half of it, Large Language Models (LLMs) were way less *large* and certainly not as *mainstream* as today. In 2022, the sudden and unexpected burst of popularity of LLMs[8] started to steer part of the research community towards SE applications for LLMs [105]. During our empirical investigations of the software documentation landscape, none of the sources we were mapping was directly mentioning any LLM or how to leverage them to obtain documentation (*e.g.,* prompt engineering for software documentation). Thus they were basically "invisible" for our landscape mapping techniques and analyses.

Nevertheless, there are a number of works addressing the use of LLMs for software documentation that we summarize in this section as a stepping stone for future research on such emerging sector of the landscape. We also discuss why LLMs are not featured among the sources.

9.4.1 LLMs for Generation of Mapped Documentation Types

One of the possible reasons behind the lack of references to LLM-based documentation sources is that these tools are often used to generate documentation types that are already mapped in the software documentation landscape. This documentation is superficially undistinguishable from the one already present in the same source.

[8]Boosted by the results achieved by OpenAI's Codex, released as beta in August 2021 (`https://openai.com/index/openai-codex/`) and GitHub Copilot (`https://github.com/features/copilot/`).

Dvivedi *et al.* compare different LLMs on the task of generating source code documentation [63]. In their study they find that LLMs, in general, outperform the original documentation on various evaluation criteria (*e.g.,* Accuracy, Completeness, Relevance, Understandability, Readability, Time Taken). Besides the LLM-based generation, the final output looks like coming from a documentation source that is already part of the landscape: Source code level documentation (inline-, function-, and class-level source code comments).[9] Moreover, the higher the level, the poorer the documentation, reinforcing the idea that it is still hard to use LLMs as higher level documentation generators, encompassing multiple files and directories in a project.

Khan *et al.* study the performance of GPT-3 across 6 programming languages for code documentation generation [124]. Codex (a GPT-3 based model pre-trained on both natural and programming languages) outperformed the state of the art even with one-shot learning.

Agarwal *et al.* evaluate three state of the art LLMs (GPT-4, GPT-3.5, and CodeLlama) for different tasks and, in particular, on method-level documentation generation in six languages (Python, Javascript, TypeScript, Java, C#, and C/C++), with respect to syntax and format correctness [3]. While GPT-4 slightly outperforms the others, in general, all models show very good syntax correctness. CodeLlama, a significantly smaller model, falls behind in TypeScript and in formal correctness of Javascript but still holds the ground, despite an apparent power disadvantage (which turns out to be an advantage in terms of cost-effectiveness).

Lin *et al.* study multi-agent systems based on LLMs to mimic the software development processes and the different roles involved [146]. In particular, the recruitment engineer agent is responsible of generating requirement documents and the software architect, from the analysis of those, generates design documents. Diggs *et al.* use LLMs to generate documentation for legacy code [59]. Their approach generates line-level comments on an Electronic Health Records (EHR) system in MUMPS[10] and open source applications in IBM mainframe Assembly Language Code.

Su *et al.* use LLMs to extract locking rules, to synthesize exception predicates, and to identify performance-related configurations from different sources (*e.g.,* source code comments, configuration manuals) [243], showing promising initial results.

Geng *et al.* investigate the ability of LLMs to understand different perspectives of code snippets and their multi-intent comment generation capabilities (*e.g.,* functionality, usage) [84]. They find Codex to be a good few shot summarizer of multi-intent code comments, outperforming the state of the art already with just 10 examples. Luo *et al.* propose RepoAgent, a tool aiming at proactively generating, maintaining, and updating repository-level code documentation [151].

In most of the previous cases, LLMs generate a form of documentation that is already present in our representation of the landscape. Instead, when the summarization of multiple artifacts and a higher level description are created, the form of documentation produced is close to human curated documentation but generated on-demand, often for program comprehension tasks.

9.4.2 LLMs for Program Comprehension and Interactive Explanation

Leinonen *et al.* compare documentation generated by LLMs with the one created by CS students [134], finding the former significantly easier to understand and more accurate. MacNeil *et al.* propose creating code summaries from code to explain code snippets in the context of CS education [152]. While being useful to compare on a small scale (*e.g.,* snippets) and to provide learning material (*e.g.,* misconceptions about CS concepts), these approaches still lack a systematic large scale approach to documentation of actual software projects.

[9] In fact, despite being mentioned in the paper, folder-level documentation is not addressed in the cited work.

[10] Massachusetts General Hospital Utility Multi-Programming System. A high-level language and database developed in the 1960s for managing patient medical records [224].

Ross *et al.* study conversational interactions with LLMs as a coding assistant for code translation, explanation, and completion on the fly [201]. Providing more context than that available to non-conversational assistants integrated in the IDE as code suggestion mechanisms, the Programmer's Assistant goes beyond mere code generation and more towards documentation generation. Nam *et al.* propose a first investigation of an LLM-based conversational UI built directly in the IDE that is geared towards code understanding [162]. Their tool, GILT, is capable of prompt-lessly (supported by the UI) explaining a highlighted section of code, providing details and usage examples of API calls, and explaining domain-specific terms, outperforming standard web searches for the same tasks. These conversational approaches ignore persistence of the generated documentation (*e.g.,* explanations).

9.4.3 Promises and Perils of LLMs for Software Documentation

The use of LLM for automatic generation of code summaries given a specific context, reminiscent of the dreams of on-demand developer documentation [193], is a promising approach that needs to overcome the limitations briefly sketched in this section before being able to scale to production level large systems, as most of those in our analyses. This is both a potential limitation of our studies and an opportunity to test further the documentation landscape's capabilities to highlight trending phenomena. By capturing specific aspects of the use of LLMs for documentation generation (*e.g.,* prompting examples, known LLMs and LLM APIs) the recent changes in the landscape could be interpreted in terms of another potential source. So far, we argue, the manifestation of these new *tools* is either not persisted or limited to being persisted in forms already known and mapped in the landscape.

The use of LLMs in software documentation generation could be a PhD thesis topic in itself. Just a word of warning, though. Very recently, industrial applications for integration of different documentation taken from the web in LLMs answers showed promising results.[11] Agentic LLMs are the new buzzword in the field [97, 246, 252]. This is an extremely fast moving target in competition with the industry, yet it enchants with the charme of so much untapped potential.

[11] February 2, 2025: OpenAI introduces the deep search feature for ChatGPT. An LLM-based agent that synthesizes large amounts of online information and completes multi-step research tasks, autonomously accessing resources on the web. See `https://openai.com/index/introducing-deep-research/`

9.5 Closing Words

Let me conclude this long trip with a recommendation I got from my advisor about my being overly formal in some contexts.[12,13] "Don't!"

We unleashed tectonic shifts, jungle overgrowths, tsunamis, and potentially all-sinking icebergs upon the software documentation. At this point in space (200+ pages in, counting everything for dramatic effect) and time (4.5 years in[14]) I have the privilege to be able to take a step back and look at my journey. While I am writing, I am probably realizing for the first time what happened in its entirety. I feel like I am really, deeply understanding it at another level now. I love this super-power of the writing process. This document tries to "briefly" and rigorously summarize the journey in a consistent account of our main findings and contributions.
But there is so much more.

As I suspect it is quite common at the end of a PhD,[15] there is a large spectrum of potential reactions. Mine is blended from an equal mix of two feelings. On the one side, there is a morale boost from putting in perspective the micro-management of all the steps. Starting from an idea discussed in front of a Red Bull® at the coffee machine and finishing with the submission of an almost pixel-perfect camera ready version of a paper. This boost is driven by the realization that the questions (even before and more than the answers) are starting to become relevant. Or at least they provide a non-negligible delta in our knowledge about software documentation and its landscape. *Luckily*, this matches with the expected outcomes of a PhD. On the opposite side there is a huge tank, filled with frustration for all the promising paths that emerged while exploring the main one. Each work that did not find enough energy,[16] that we failed at pushing and transforming from idea to publication, each and every one of these works contributes to filling that tank. Probably this is also an expected outcome of a PhD, a tougher one to look at.

I strongly believe in publishing, not to avoid perishing, but to distill months (sometimes too many months) of work in a 6-to-666 minutes reading that, more than informing, can inspire other human beings[17] to pursue similar *or totally different* research ideas, with the right mix of curiosity, ingenuity, expertise, scientific rigor, hard work, smart shortcuts, and creativity. I did not find the perfect balance yet, but I try to get closer every day.[18]

I will not pretend to have changed the future of humankind with my research, the one summarized in this thesis, nor even counting the one that did not fit in this document. Nevertheless, I enjoyed the process and *the processors*, and much is yet to be done, investigated, and written. You can stop reading here, I am eagerly looking forward to the next chapter.

[12] To be pronounced half laughing, half being damn serious, as only some people can do effectively.

[13] The following 42 millimeters of *hspace* have been purposefully reserved for visual emphasis representing a dramatic pause.

[14] The year count is just an underestimation, considering my previous experience at the Biorobotics Institute, that surely influenced my approach and my resilience.

[15] With no scientific evidence, nor the intention to look for it, looking at you Andi!

[16] From the aforementioned Red Bulls, or even cheaper clones.

[17] I will extend this definition in my mental model of reality soon enough.

[18] A final footnote before concluding. Please, read the Bartimaeus trilogy by Jonathan Stroud, or at least have a look at the beginning of The Amulet of Samarkand. If you ignore the debatable "young adult" labeling of the books' genre, besides the quite unique fantasy setting, Stroud's layered use of footnotes is simply amazing.

Appendices

A

RagnaDok Implementation

We present the architecture and implementation of RagnaDok, a tool to visualize the documentation landscape that was used to devise the taxonomy presented in Chapter 3. Based on the taxonomy, the tool evolved as a front-end to explore and visualize the documentation landscape. RagnaDok has been implemented by Tommaso Rodolfo Masera as part of his Master Thesis [196], which I co-supervised.

A.1 System Architecture

RagnaDok is a web-based application built using the React.js library for its frontend and Flask for its Python backend. It is currently deployed at `https://ragnadok.si.usi.ch/`.

The system is split into two main parts (Figure A.1), the backend and the frontend.

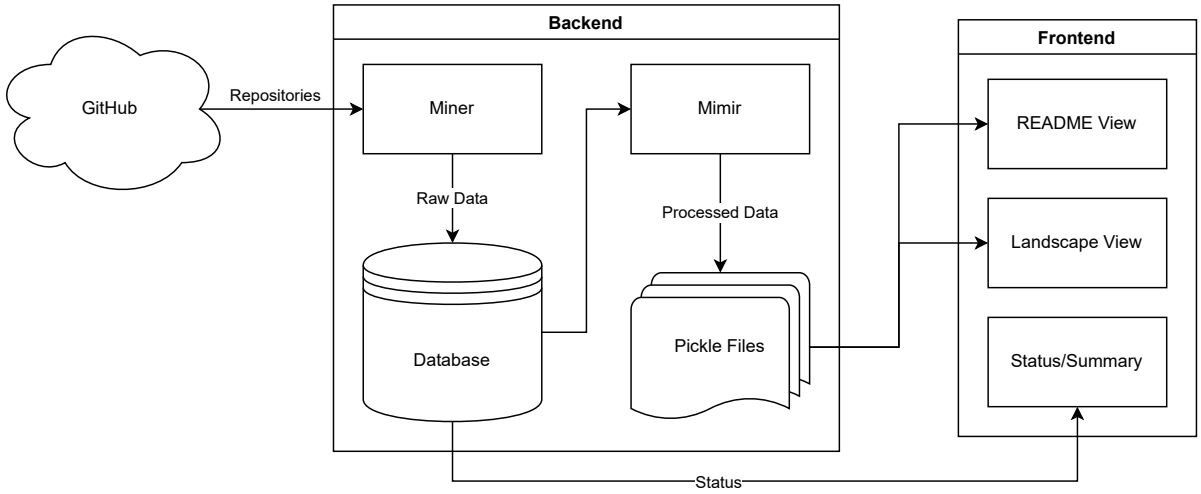


Figure A.1: Architecture of RagnaDok. Image taken from the Master Thesis of Tommaso Rodolfo Masera (co-supervised) [196].

It depends on GitHub externally to mine data. The backend takes care of mining data, processing it and sending the results to the frontend. The frontend requests the data from the backend and displays it in various visualizations in a web browser. The mining process is split into two parts. Raw, unprocessed data is written to a SQLite database, which is also used as a synchronizing point for the status of the mining.

Then Mimir, the historian, receives the raw data and processes it, writing it to a pickle file to store itself and its content in a binary file. The pickle files are then un-pickled when their specific repository is requested by the frontend and they supply the requested data to its view.

A.2 Data and History Mining

The main actions that *git* tracks in a commit are: Add, Modify, Delete, Rename, Move (see Section 3.2.1). The miner begins by acquiring the software repository that we wish to examine. To get access to the repository we can use the GitHub API or clone the repository.

GitHub APIs have a limit of 5,000 requests per hour and our objective is to mine multiple repositories concurrently. In an exploratory study, we found that a single large project could require a large number of requests by itself, making this approach unfeasible at the required scale. Between these two approaches, we thus choose a local deep[1] clone. With a deep clone, we can extract information regarding all the actions that were performed on all files without any requests at all, thus avoiding GitHub rate limits.

A.2.1 Using the Log

git maintains a log of the commits performed in a repository. We run the `git log` command with some custom flags in order to have a formatted output of the information about commits that we can parse. The flags that we employ are the following:

- `-C [path]`: takes a `path` argument and tells *git* to use it to run commands. We pass the absolute path to the cloned repository. This flag is actually a flag for the `git` command and not for the `git log` command, so it must come before `log`.
- `--name-status`: displays a letter representing the action performed on the files that were modified in a commit, at the start of the line (*e.g.,* `A` for additions, `M` for modifications). With this option, renames are shown in the log with `R[N]`, with `[N]`$\in \{0, 1, ..., 100\}$, representing *git*'s confidence that the file has been renamed rather than being a new file.
- `--pretty=tformat:[format]`: Allows us to format other information in the commit (*e.g.,* date, author, commit hash). The custom format we use is: "`CommitHash:%H%nDate:%ad%nCommitAuthor:%an%nCommitMessage:%B`", to display the hash of the commit, its date, and its message.
- `--date=unix`: Formats all the date as a Unix timestamp to avoid inconsistencies.

In Figure A.2, we show an example output with the selected flags.

A.2.2 Filtering Commits

The custom formatted output of the `git log` command contains all files that have been affected by each commit. We can now filter all the existing commits based on whether they modified a README file. To do so, we need to define what classifies as a README file. We normalize the filename to lower case and use a heuristic-based approach to filtering.

[1]The complete history is cloned, compared to a shallow clone where only the latest snapshot is cloned locally.

```
CommitHash: 6eabfa27e7c380ed762f088092ee89d4bcc0b91a
Date: 1688314939
CommitAuthor: John Git
CommitMessage: add super cool feature

A  NewFile.java
R100  old_file.txt new_file.txt
D  deleted_file.py
```

Figure A.2: Example of commit obtained from the `git log` command with the selected flags.

A file is a README if:

- Its filename contains one of the following sub-strings as whole words: `readme`, `infotext`, `read me`, `read.me`, `read_me`, `read-me`. A whole word in our case is a sub-string that matches one of the given sub-strings that is not bounded by any other alphabetical character. For example, `this_is_a_readme_file.txt` is considered as a README file: The sub-string `readme` is a whole word bounded by underscore characters. Instead, `ReadMessage` will not be included as, while the sub-string `readme` is present, it is not a whole word.
- Its filename does not have any of the following extensions: `.py`, `.java`, `.c`, `.cpp`, `.cc`, `.js`, `.jsx`, `.png`, `.jpg`, `.jpeg`, `.gif`, `.svg`, `.ico`, `.ino`. This is to avoid files that contain the sub-string, but that are not text files according to their extensions.

Heuristic-based approaches to README identification have clear limitations. Although the case of `ThisIsAReadme.txt` will not be captured by our heuristics, this should have a limited impact on our results. From manual inspection, a typical README file does not have a very complicated or long name. In such a case, we prefer to ignore complex filenames in favor of simpler and more accurate ones. When the heuristic is satisfied for any single affected file in a commit, we mark this commit as a README commit to check out during the next step of the mining. We always include the first and last commits made on a repository regardless of their modified files as they represent the start and end of the whole history of the repository.

A.2.3 Building Histories

With information on all the relevant commits, we can build the README histories of the project. We perform a `git checkout` at each identified commit and we check all the modified files. For each modified file, we check whether the heuristics for README files are satisfied. If there is a modified readme, we create a new README history or we update a previously existing one, and we append to it a new README version which contains all the relevant information.

For each commit, we use the `scc`[2] command line tool to count the lines of comments in the repository, as source code comments are a type of documentation source.[3] In addition, we separately keep track of commits that deleted or renamed a README file to determine when a history ends and to simplify history chaining (see Section 3.2.3). Once the raw history data is mined, we store it into an SQLite database and process it.

[2]See `https://github.com/boyter/scc`

[3]We compared the performance of `scc` with the better known `cloc` [333] and `scc` provided faster outputs with a negligible impact in accuracy.

B

Detection Methods for Taxonomy Categories

We present, for each leaf category in the preliminary taxonomy of the documentation landscape (Section 3.3), the detection methods used by RagnaDok to capture links to documentation sources of the specified category or sub-category.

B.1 Blog

B.1.1 Custom Blog

Detection method: URL contains the sub-string `blog`.

B.1.2 Medium

Detection method: Top-level domain contains the sub-string `medium`.

B.1.3 WordPress

Detection method: Top-level domain contains any of the following sub-strings: `wordpress`, `wp.me`, `wp.com`.

B.2 Community Platform

B.2.1 Custom Community Platform

Detection method: URL contains the word `community`.

B.2.2 Instant Messaging

Discord

Detection method: Top-level domain begins with any of `https://discord.gg` or `https://discord.com/invite`, or contains the sub-string `discord`.

Gitter

Detection method: Top-level domain begins with `https://gitter.im`, or contains the word `gitter`.

IRC

Detection method: Contains any of the following whole words: `irc, oftc, freenode`.

Slack

Detection method: Top-level domain contains the sub-string `slack`.

Telegram

Detection method: URL Contains any of the following sub-strings: `telegram, t.me`.

B.2.3 Media Sharing

Imgur

Detection method: Top-level domain contains the sub-string `imgur`.

Vimeo

Detection method: Top-level domain contains the sub-string `vimeo`.

YouTube

Detection method: Top-level domain contains any of the following sub-strings: `youtube, youtu.be`.

B.2.4 Social Media

Facebook

Detection method: Top-level domain contains the sub-string `facebook`.

Instagram

Detection method: Top-level domain contains the sub-string `instagram`.

TikTok

Detection method: Top-level domain contains the sub-string `tiktok`.

Twitter

Detection method: Top-level domain contains the sub-string `twitter`.[1]

[1]Note that at the time of this study, the X domain was not yet available, the company was in the process of rebranding and the migration to "X.com" was completed in May 2024 [334].

B.3 Document

B.3.1 Multimedia Document

Audio

Detection method: The URL ends with a dot (.) followed by any of these file extensions: `mp3, wav, wma, aac, flac, alac, ogg, aiff, dsd, pcm, aif, mp2, m4a, m4b, m4p, m4r, mid, midi, mka, mpa, ra, ram, tta, wv, wvp, 3gp, aa, aax, act, aifc, amr, ape, awb, dct, dss, dvf, gsm, iklax, ivs, m3u, m3u8, m4r, mmf, mpc, msv, oga, opus, ra, rm, sln, vox, w64, wma, wv, xspf, 8svx`.

Image

Detection method: The URL ends with a dot (.) followed by any of these file extensions: `png, jpg, jpeg, gif, bmp, svg, tiff, tif, eps, raw, cr2, nef, orf, sr2, webp, heif, heic`.

Video

Detection method: The source must match detection for YouTube and the URL must contain `watch?v=` as the link format for YouTube videos, or it must match Vimeo and must contain a video ID according to the regular expression `d+`, or it must end with a dot (.) followed by any of the following file extensions: `mp4, mov, avi, wmv, flv, mkv, webm, m4v, mpeg, mpg`.

B.3.2 Textual Document

Book

Detection method: Ends with any of the following: `[.epub, .mobi]` or contains any of the following sub-strings: `amazon, barnesandnoble, kobo, goodreads, abebooks, bookdepository, booktopia, indiebound, alibris, thriftbooks, booksamillion, bookshop, book-finder, bookish, bookbub, bookriot, bookpage, bookforum, bookreporter, bookbrowse`.

Text File

Detection method: Does not contain any of the following words: `index, home, main, default, welcome, landing, start, base, front` and ends with any of the following extensions: `txt, md, rst, xml, json, yaml, yml, csv, tsv, tex, rtf, doc, docx, odt, pdf, djvu, fb2, xps, cbz, cbr, cb7, cbt, cba, chm, pdb, prc, azw, azw3, lit, ps, pml, htmlz, txtz, rtfz, pdfz`.

B.4 Forum

Custom Forum

Detection method: URL contains any of the following sub-strings: `forum, discuss` and does not satisfy GitHub Discussions.

B.4.1 GitHub Discussions

Detection method: Top-level domain contains `github.com` and URL contains `discussions`.

B.4.2 StackOverflow

Detection method: Top-level domain contains the sub-string `stackoverflow`.

B.4.3 Homepage

Project Homepage

Detection method: Given the name of the GitHub project it belongs to in the form of `<owner>/<name>`, the URL contains either `<owner>` or `<name>`.

Third-Party Project Homepage

Detection method: Given the name of the GitHub project it belongs to in the form of `<owner>/<name>`, the URL contains neither of them and contains any top-level domain from the Internet Assigned Numbers Authority (IANA).[2]

Specific Subsection

Detection method: We remove `https://` or `http://` from the URL if it exists. Then, we identify three cases:

1. One or less occurrences of the `/` character in the URL: If there are no `/` characters in the URL, then there is no deeper path that is referenced. A specific subsection must have at least one occurrence of `/`, and the part after the first `/` must be non-empty. For instance:

   - `example.com`[/]: Not a specific subsection.
   - `example.com/<some_sub_path_here>`: Specific subsection.

2. One or two `/` characters in the URL, and the part following the second occurrence of `/` must be empty. Here, we also take into consideration some common words that can appear in the URL of a homepage: `index, home, main, default, welcome, landing, start, root, base, front`. We found homepages linked as: `example.com/index.html/`. While this apparently has a second level path, indicating a potential subsection, it actually refers to a homepage. Therefore, a URL that has between one and two occurrences of the `/` character is a specific subsection if all of the following conditions are satisfied:

   - if there is only one occurrence of `/`, the part of the string following the `/` is non-empty (*e.g.,* `example.com/<some_sub_path_here>`)
   - if there are two `/` characters, the URL ends with the second occurrence of `/`
   - the content following the first occurrence of `/` does not contain any of the aforementioned "homepage words"

3. More than two occurrences of `/` or the URL contains a # character. A number of slashes indicates a deeper path and the # character indicates a specific section referenced typically in HTML or Markdown documents through anchor points.

[2] See `https://data.iana.org/TLD/tlds-alpha-by-domain.txt`

B.5 Mailing List

B.5.1 Custom Mailing List

Detection method: URL contains as sub-string any of `list, mailing, mail` and does not satisfy neither Google Groups nor Mailman.

B.5.2 Google Groups

Detection method: Top-level domain contains the sub-string `groups.google`.

B.5.3 Mailman

Detection method: URL contains the sub-string `mailman`.

B.6 Repository Related

B.6.1 Bug Tracker

Bugzilla

Detection method: URL contains the sub-string `bugzilla`.

Jira

Detection method: URL contains the sub-string `jira`.

B.6.2 Issue Tracker

GitHub Issue Tracker

Detection method: Top-level domain contains `github.com` and URL contains the sub-string `issues`.

B.6.3 Pull Requests

GitHub Pull Request

Detection method: Top-level domain contains `github.com` and URL contains the `pull` sub-string.

B.6.4 Relative File

Detection method: the URL contains both sub-strings `github` and `/blob/` and this source does not satisfy the Source File or Textual Document heuristics. A relative path in a README file is not a full URL (*e.g.,* `../../filename.extension`). We must reconstruct this URL as GitHub represents specific files at specific commits. We check if this file exists in the reconstructed URL.

B.6.5 Repository

Bitbucket

Detection method: Contains the sub-string `bitbucket`.

GitHub

Detection method: Contains the sub-string `github.com` and does not detect any of GitHub Issues, GitHub Discussions, GitHub Wiki, and Relative File.

Launchpad

Detection method: Top-level domain contains the sub-string `launchpad.net`.

SourceForge

Detection method: Top-level domain contains the sub-string `sourceforge.net`.

Weblate

Detection method: Top-level domain contains the sub-string `weblate.org`.

B.6.6 Source File

Detection method: Ends with any of the known language extensions: `[py, java, c, cpp, cs, js, ts, php, css, scss, less, xml, json, yaml, cc, h, hpp, hxx, hh, yml, sh, bat, cmd, ps1, psm1, psd1, ps1xml, pssc, vbs, vba, vb, bas, frm, cls, ctl, xslt, xsd, wsf, wsc, wsh, ini, inf, reg, cfg, config, conf, properties, prop, props, sln, csproj, vbproj, vcxproj, vcproj, vcproj, xcodeproj, dproj, cbproj, pbxproj, pbproj, xib, storyboard, plist, nib]`.

B.7 Wiki

B.7.1 Custom Wiki

Detection method: Contains the sub-string `wiki` and does not satisfy Wikipedia or GitHub Wiki.

B.7.2 GitHub Wiki

Detection method: Top-level domain contains `github.com` and URL contains `wiki`.

B.7.3 Wikipedia

Detection method: Top-level domain contains the sub-string `wikipedia`.

Bibliography

[1] Abreu, R. and Premraj, R. [2009]. How developer communication frequency relates to bug introducing changes, *Proceedings of IWPSE-EVOL 2009 (Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops)*, ACM, pp. 153–158.

[2] Adams, P. H. and Martell, C. H. [2008]. Topic detection and extraction in chat, *Proceedings of ICSC 2008 (International Conference on Semantic Computing)*, IEEE, pp. 581–588.

[3] Agarwal, A., Chan, A., Chandel, S., Jang, J., Miller, S., Moghaddam, R. Z., Mohylevskyy, Y., Sundaresan, N. and Tufano, M. [2024]. Copilot evaluation harness: Evaluating LLM-guided software programming, arxiv:2402.14261.
URL: *https://arxiv.org/abs/2402.14261*

[4] Aghajani, E., Bavota, G., Linares-Vásquez, M. and Lanza, M. [2021]. Automated documentation of Android apps, *IEEE Transactions on Software Engineering* **47**(1): 204–220.

[5] Aghajani, E., Nagy, C., Linares-Vásquez, M., Moreno, L., Bavota, G., Lanza, M. and Shepherd, D. C. [2020]. Software documentation: The practitioners' perspective, *Proceedings of ICSE 2020 (International Conference on Software Engineering)*, ACM, pp. 590–601.

[6] Aghajani, E., Nagy, C., Vega-Márquez, O. L., Linares-Vásquez, M., Moreno, L., Bavota, G. and Lanza, M. [2019]. Software documentation issues unveiled, *Proceedings of ICSE 2019 (International Conference on Software Engineering)*, IEEE, pp. 1199–1210.

[7] Alhir, S. S. [1998]. *UML in a Nutshell*, O'Reilly.

[8] Ali, N., Baker, S., O'Crowley, R., Herold, S. and Buckley, J. [2018]. Architecture consistency: State of the practice, challenges and requirements, *Empirical Software Engineering* **23**(1): 224–258.

[9] Alkadhi, R., Lata, T., Guzmany, E. and Bruegge, B. [2017]. Rationale in development chat messages: An exploratory study, *Proceedings of MSR 2017 (International Conference on Mining Software Repositories)*, IEEE/ACM, pp. 436–446.

[10] Aniche, M., Treude, C., Steinmacher, I., Wiese, I., Pinto, G., Storey, M.-A. and Gerosa, M. A. [2018]. How modern news aggregators help development communities shape and share knowledge, *Proceedings of ICSE 2018 (International Conference on Software Engineering)*, ACM, pp. 499–510.

[11] Arisholm, E., Briand, L. C., Hove, S. E. and Labiche, Y. [2006]. The impact of UML documentation on software maintenance: An experimental evaluation, *IEEE Transactions on Software Engineering* **32**(6): 365–381.

[12] Arya, D. M., Guo, J. L. C. and Robillard, M. P. [2024]. Properties and styles of software technology tutorials, *IEEE Transactions on Software Engineering* **50**(2): 159–172.

[13] Arya, D. M., Nassif, M. and Robillard, M. P. [2022]. A data-centric study of software tutorial design, *IEEE Software* **39**(3): 106–115.

[14] Assunção, W. K. G., Vergilio, S. R. and Lopez-Herrejon, R. E. [2023a]. *ModelVars2SPL: From UML Class Diagram Variants to Software Product Line Core Assets*, Springer, chapter in Handbook of Re-Engineering Software Intensive Systems into Software Product Lines, pp. 221–250.

[15] Assunção, W. K. G., Vergilio, S. R. and Lopez-Herrejon, R. E. [2023b]. *Reengineering UML Class Diagram Variants into a Product Line Architecture*, Springer, chapter in UML-Based Software Product Line Engineering with SMarty, pp. 393–414.

[16] Baabad, A., Zulzalil, H. B., Hassan, S. and Baharom, S. B. [2022]. Characterizing the architectural erosion metrics: A systematic mapping study, *IEEE Access* **10**: 22915–22940.

[17] Bacchelli, A. and Bird, C. [2013]. Expectations, outcomes, and challenges of modern code review, *Proceedings of ICSE 2013 (International Conference on Software Engineering)*, IEEE, pp. 712–721.

[18] Bacchelli, A., Dal Sasso, T., D'Ambros, M. and Lanza, M. [2012]. Content classification of development emails, *Proceedings of ICSE 2012 (International Conference on Software Engineering)*, IEEE, pp. 375–385.

[19] Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Herrick, H. L., Hughes, R. A., Mitchell, L. B., Nelson, R. A., Nutt, R., Sayre, D., Sheridan, P. B., Stern, H. and Ziller, I. [1956]. Fortran: Automatic coding system for the IBM 704 EDPM.

[20] Badreddin, O., Lethbridge, T. C. and Elassar, M. [2013]. Modeling practices in open source software, *Proceedings of OSS 2013 (Open Source Software: Quality Verification)*, Springer, pp. 127–139.

[21] Bajaj, K., Pattabiraman, K. and Mesbah, A. [2014]. Mining questions asked by web developers, *Proceedings of MSR 2014 (Working Conference on Mining Software Repositories)*, ACM, pp. 112–121.

[22] Baltes, S. and Diehl, S. [2014]. Sketches and diagrams in practice, *Proceedings of FSE 2014 (International Symposium on Foundations of Software Engineering)*, ACM, pp. 530–541.

[23] Baltes, S. and Treude, C. [2020]. Code duplication on Stack Overflow, *Proceedings of ICSE 2020 (International Conference on Software Engineering)*, IEEE, pp. 13–16.

[24] Balzer, M. and Deussen, O. [2005]. Exploring relations within software systems using treemap enhanced hierarchical graphs, *Proceedings of VISSOFT 2005 (International Workshop on Visualizing Software for Understanding and Analysis)*, IEEE, pp. 1–6.

[25] Balzer, M., Deussen, O. and Lewerentz, C. [2005]. Voronoi treemaps for the visualization of software metrics, *Proceedings of SoftVis 2005 (Symposium on Software Visualization)*, ACM, pp. 165–172.

[26] Bavota, G., Canfora, G., Di Penta, M., Oliveto, R. and Panichella, S. [2013]. An empirical investigation on documentation usage patterns in maintenance tasks, *Proceedings of ICSM 2013 (International Conference on Software Maintenance)*, IEEE, pp. 210–219.

[27] Beck, K. [2000]. *Extreme Programming Explained: Embrace Change*, Addison-Wesley Professional.

[28] Behutiye, W., Seppänen, P., Rodríguez, P. and Oivo, M. [2020]. Documentation of quality requirements in agile software development, *Proceedings of EASE 2020 (Evaluation and Assessment in Software Engineering)*, ACM, pp. 250–259.

[29] Bergel, A. [2022]. *Agile Visualization with Pharo – Crafting Interactive Visual Support Using Roassal*, Apress, Berkeley, CA.

[30] Bernardi, M. L., Canfora, G., Di Lucca, G. A., Di Penta, M. and Distante, D. [2012]. Do developers introduce bugs when they do not communicate? The case of Eclipse and Mozilla, *Proceedings of CSMR 2012 (European Conference on Software Maintenance and Reengineering)*, IEEE, pp. 139–148.

[31] Beyer, S., Macho, C., Pinzger, M. and Di Penta, M. [2018]. Automatically classifying posts into question categories on Stack Overflow, *Proceedings ICPC 2018 (International Conference on Program Comprehension)*, ACM, pp. 211–221.

[32] Bird, C., Gourley, A., Devanbu, P., Gertz, M. and Swaminathan, A. [2006]. Mining email social networks, *Proceedings of MSR 2006 (International Workshop on Mining Software Repositories)*, ACM, pp. 137–143.

[33] Boehm, B. W. and Papaccio, P. N. [1988]. Understanding and controlling software costs, *IEEE Transactions on Software Engineering* **14**(10): 1462–1477.

[34] Booch, G. [1993]. *Object-Oriented Analysis and Design with Applications*, 2nd edn, Benjamin-Cummings Publishing Co., Inc.

[35] Borges, H. and Valente, M. T. [2018]. What's in a GitHub star? Understanding repository starring practices in a social coding platform, *Journal of Systems and Software* **146**: 112–129.

[36] Brooks, R. [1978]. Using a behavioral theory of program comprehension in software engineering, *Proceedings of ICSE 1978 (International Conference on Software Engineering)*, IEEE, pp. 196–201.

[37] Bruneliere, H., Cabot, J., Dupé, G. and Madiot, F. [2014]. Modisco: A model driven reverse engineering framework, *Information and Software Technology* **56**(8): 1012–1032.

[38] Campanella, S. and Lanza, M. [2024]. Hidden in the code: Visualizing true developer identities, *Proceedings of VISSOFT 2024 (Working Conference on Software Visualization)*, IEEE, pp. 24–35.

[39] Campbell, J. C., Zhang, C., Xu, Z., Hindle, A. and Miller, J. [2013]. Deficient documentation detection — A methodology to locate deficient project documentation using topic analysis, *Proceedings of MSR 2013 (Working Conference on Mining Software Repositories)*, IEEE, pp. 57–60.

[40] Carruthers, S., Thomas, A., Kaufman-Willis, L. and Wang, A. [2023]. Growing an accessible and inclusive systems design course with PlantUML, *Proceedings of SIGCSE 2023 (Technical Symposium on Computer Science Education)*, ACM, pp. 249–255.

[41] Cerezo, J., Kubelka, J., Robbes, R. and Bergel, A. [2019]. Building an expert recommender chatbot, *Proceedings of BotSE 2019 (International Workshop on Bots in Software Engineering)*, IEEE/ACM, pp. 59–63.

[42] Chatterjee, P., Damevski, K., Kraft, N. A. and Pollock, L. [2020]. Software-related Slack chats with disentangled conversations, *Proceedings of MSR 2020 (International Conference on Mining Software Repositories)*, ACM, pp. 588–592.

[43] Chatterjee, P., Damevski, K., Pollock, L., Augustine, V. and Kraft, N. A. [2019]. Exploratory study of Slack Q&A chats as a mining source for software engineering tools, *Proceedings of MSR 2019 (International Conference on Mining Software Repositories)*, IEEE, pp. 490–501.

[44] Chatterjee, P., Kong, M. and Pollock, L. [2020]. Finding help with programming errors: An exploratory study of novice software engineers' focus in Stack Overflow posts, *Journal of Systems and Software* **159**: 110454.

[45] Chen, J.-C. and Huang, S.-J. [2009]. An empirical analysis of the impact of software development problem factors on software maintainability, *Journal of Systems and Software* **82**(6): 981–992.

[46] Chen, Q., Grundy, J. and Hosking, J. [2003]. An e-whiteboard application to support early design-stage sketching of UML diagrams, *Proceedings of HCC 2003 (Symposium on Human Centric Computing Languages and Environments)*, IEEE, pp. 219–226.

[47] Chowdhury, S. A. and Hindle, A. [2015]. Mining StackOverflow to filter out off-topic IRC discussion, *Proceedings of MSR 2015 (Working Conference on Mining Software Repositories)*, IEEE, pp. 422–425.

[48] Constantino, K., Zhou, S., Souza, M., Figueiredo, E. and Kästner, C. [2020]. Understanding collaborative software development: An interview study, *Proceedings of ICGSE 2020 (International Conference on Global Software Engineering)*, ACM, pp. 55–65.

[49] Costa Silva, C., Galster, M. and Gilson, F. [2022]. A qualitative analysis of themes in instant messaging communication of software developers, *Journal of Systems and Software* **192**: 111397.

[50] Costa Silva, C., Galster, M. and Gilson, F. [2024]. Applying short text topic models to instant messaging communication of software developers, *Journal of Systems and Software* **216**: 112111.

[51] Costa Silva, C. M. [2020]. Reusing software engineering knowledge from developer communication, *Proceedings of ESEC/FSE (European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, pp. 1682–1685.

[52] Dabic, O., Aghajani, E. and Bavota, G. [2021]. Sampling projects in GitHub for MSR studies, *Proceedings of MSR 2021 (International Conference on Mining Software Repositories)*, IEEE/ACM, pp. 560–564.

[53] Dachselt, R., Frisch, M. and Decker, E. [2008]. Enhancing UML sketch tools with digital pens and paper, *Proceedings of SoftVis 2008 (Symposium on Software Visualization)*, ACM, pp. 203–204.

[54] Dagenais, B. and Robillard, M. P. [2010]. Creating and evolving developer documentation: Understanding the decisions of open source contributors, *Proceedings of FSE 2010 (International Symposium on Foundations of Software Engineering)*, ACM, pp. 127–136.

[55] de Silva, L. and Balasubramaniam, D. [2012]. Controlling software architecture erosion: A survey, *Journal of Systems and Software* **85**(1): 132–151.

[56] De Vito, G., Ferrucci, F. and Gravino, C. [2020]. Design and automation of a COSMIC measurement procedure based on UML models, *Software and Systems Modeling* **19**(1): 171–198.

[57] Di Sorbo, A., Panichella, S., Visaggio, C. A., Di Penta, M., Canfora, G. and Gall, H. C. [2015]. Development emails content analyzer: Intention mining in developer discussions, *Proceedings of ASE 2015 (International Conference on Automated Software Engineering)*, IEEE, pp. 12–23.

[58] Di Sorbo, A., Panichella, S., Visaggio, C. A., Penta, M. D., Canfora, G. and Gall, H. C. [2021]. Exploiting natural language structures in software informal documentation, *IEEE Transactions on Software Engineering* **47**(8): 1587–1604.

[59] Diggs, C., Doyle, M., Madan, A., Scott, S., Escamilla, E., Zimmer, J., Nekoo, N., Ursino, P., Bartholf, M., Robin, Z., Patel, A., Glasz, C., Macke, W., Kirk, P., Phillips, J., Sridharan, A., Wendt, D., Rosen, S., Naik, N., Brunelle, J. F. and Thaker, S. [2024]. Leveraging LLMs for legacy code modernization: Challenges and opportunities for LLM-generated documentation, arxiv:2411.14971.
URL: *https://arxiv.org/abs/2411.14971*

[60] Dijkstra, E. W. [1972]. The humble programmer, *Communications of the ACM* **15**(10): 859–866.

[61] Ding, W., Liang, P., Tang, A. and van Vliet, H. [2014]. Knowledge-based approaches in software documentation: A systematic literature review, *Information and Software Technology* **56**(6): 545–567.

[62] Ducasse, S. and Pollet, D. [2009]. Software architecture reconstruction: A process-oriented taxonomy, *IEEE Transactions on Software Engineering* **35**(4): 573–591.

[63] Dvivedi, S. S., Vijay, V., Pujari, S. L. R., Lodh, S. and Kumar, D. [2024]. A comparative analysis of large language models for code documentation generation, *Proceedings of AIware 2024 (International Conference on AI-Powered Software)*, ACM, pp. 65–73.
URL: *https://doi.org/10.1145/3664646.3664765*

[64] Dzidek, W. J., Arisholm, E. and Briand, L. C. [2008]. A realistic empirical evaluation of the costs and benefits of UML in software maintenance, *Transactions on Software Engineering* **34**(3): 407–432.

[65] Ebert, V., Graziotin, D. and Wagner, S. [2022]. How are communication channels on GitHub presented to their intended audience? — A thematic analysis, *Proceedings of EASE 2022 (International Conference on Evaluation and Assessment in Software Engineering)*, ACM, pp. 40–49.

[66] Ehsan, O., Hassan, S., Mezouar, M. E. and Zou, Y. [2020]. An empirical study of developer discussions in the Gitter platform, *Transactions on Software Engineering and Methodology* **30**(1): 1–39.

[67] Elshoff, J. L. [1976]. An analysis of some commercial PL/I programs, *IEEE Transactions on Software Engineering* **SE-2**(2): 113–120.

[68] Elsner, M. and Charniak, E. [2008]. You talking to me? A corpus and algorithm for conversation disentanglement, *Proceedings of ACL-HLT 2008 (Association for Computational Linguistics: Human Language Technologies)*, ACL, pp. 834–842.

[69] Elsner, M. and Charniak, E. [2010]. Disentangling chat, *Computational Linguistics* **36**(3): 389–409.

[70] Endres, A. [1996]. A synopsis of software engineering history: The industrial perspective, *History of Software Engineering* pp. 20–24.

[71] Engels, G., Hausmann, J. H., Lohmann, M. and Sauer, S. [2006]. Teaching UML is teaching software engineering is teaching abstraction, *Proceedings of MoDELS-Satellite Events 2005 (Educator's Symposium at the International Conference on Model Driven Engineering Languages and Systems)*, Springer, pp. 306–319.

[72] Estler, H.-C., Nordio, M., Furia, C. A., Meyer, B. and Schneider, J. [2014]. Agile vs. structured distributed software development: A case study, *Empirical Software Engineering* **19**(5): 1197–1224.

[73] Faccin Vernier, E., Telea, A. C. and Comba, J. [2018]. Quantitative comparison of dynamic treemaps for software evolution visualization, *Proceedings of VISSOFT 2018 (Working Conference on Software Visualization)*, IEEE, pp. 96–106.

[74] Fagan, M. E. [1976]. Design and code inspections to reduce errors in program development, *IBM Systems Journal* **15**(3): 182–211.

[75] Favre, J.-M. [1997]. Understanding-in-the-large, *Proceedings of IWPC 1997 (International Workshop on Program Comprehension)*, IEEE, pp. 29–38.

[76] Favre, L. [2008]. Formalizing MDA-based reverse engineering processes, *Proceedings of SERA 2008 (International Conference on Software Engineering Research, Management and Applications)*, IEEE, pp. 153–160.

[77] Fernández-Sáez, A. M., Caivano, D., Genero, M. and Chaudron, M. R. V. [2015]. On the use of UML documentation in software maintenance: Results from a survey in industry, *Proceedings of MODELS 2015 (International Conference on Model Driven Engineering Languages and Systems)*, IEEE, pp. 292–301.

[78] Forward, A., Badreddin, O. and Lethbridge, T. C. [2010]. Perceptions of software modeling: A survey of software practitioners, *Proceedings of C2M: EEMDD 2010 (Workshop from Code Centric to Model Centric: Evaluating the Effectiveness of Model Driven Development)*.

[79] Forward, A. and Lethbridge, T. C. [2002]. The relevance of software documentation, tools and technologies: A survey, *Proceedings of DocEng 2002 (Symposium on Document Engineering)*, ACM, pp. 26–33.

[80] Foundjem, A. and Adams, B. [2021]. Release synchronization in software ecosystems, *Empirical Software Engineering* **26**(3): 34.

[81] Fowler, M. [2018]. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd edn, Addison-Wesley.

[82] Fowler, M., Highsmith, J. et al. [2001]. The agile manifesto, *Software Development* **9**(8): 28–35.

[83] Garousi, G., Garousi-Yusifoğlu, V., Ruhe, G., Zhi, J., Moussavi, M. and Smith, B. [2015]. Usage and usefulness of technical software documentation: An industrial case study, *Information and Software Technology* **57**: 664–682.

[84] Geng, M., Wang, S., Dong, D., Wang, H., Li, G., Jin, Z., Mao, X. and Liao, X. [2023]. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning, arXiv:2304.11384.
URL: *https://arxiv.org/abs/2304.11384*

[85] Geremia, S., Bavota, G., Oliveto, R., Lanza, M. and Di Penta, M. [2019]. Characterizing leveraged Stack Overflow posts, *Proceedings of SCAM 2019 (International Working Conference on Source Code Analysis and Manipulation)*, IEEE, pp. 141–151.

[86] German, D. M., Adams, B. and Hassan, A. E. [2013]. The evolution of the R software ecosystem, *Proceedings of CSMR 2013 (European Conference on Software Maintenance and Reengineering)*, IEEE, pp. 243–252.

[87] Goeminne, M. and Mens, T. [2010]. A framework for analysing and visualising open source software ecosystems, *Proceedings of IWPSE-EVOL 2010 (Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops)*, ACM, pp. 42–47.

[88] Goeminne, M. and Mens, T. [2013]. A comparison of identity merge algorithms for software repositories, *Science of Computer Programming* **78**(8): 971–986.

[89] Gote, C. and Zingg, C. [2021]. gambit — An open source name disambiguation tool for version control systems, *Proceedings of MSR 2021 (International Conference on Mining Software Repositories)*, IEEE, pp. 80–84.

[90] Gravino, C., Scanniello, G. and Tortora, G. [2015]. Source-code comprehension tasks supported by UML design models: Results from a controlled experiment and a differentiated replication, *Journal of Visual Languages and Computing* **28**: 23–38.

[91] Guzman, E., Alkadhi, R. and Seyff, N. [2016]. A needle in a haystack: What do Twitter users say about software?, *Proceedings of RE 2016 (International Requirements Engineering Conference)*, IEEE, pp. 96–105.

[92] Guzzi, A., Bacchelli, A., Lanza, M., Pinzger, M. and Van Deursen, A. [2013]. Communication in open source software development mailing lists, *Proceedings of MSR 2013 (Working Conference on Mining Software Repositories)*, IEEE, pp. 277–286.

[93] Habib, B. and Romli, R. [2021]. A systematic mapping study on issues and importance of documentation in agile, *Proceedings of ICSESS 2021 (International Conference on Software Engineering and Service Science)*, IEEE, pp. 198–202.

[94] Hadar, I., Sherman, S., Hadar, E. and Harrison, J. J. [2013]. Less is more: Architecture documentation for agile development, *Proceedings of CHASE 2013 (International Workshop on Cooperative and Human Aspects of Software Engineering)*, IEEE, pp. 121–124.

[95] Hata, H., Novielli, N., Baltes, S., Kula, R. G. and Treude, C. [2022]. GitHub Discussions: An exploratory study of early adoption, *Empirical Software Engineering* **27**(1): 1–32.

[96] Hata, H., Treude, C., Kula, R. G. and Ishio, T. [2019]. 9.6 million links in source code comments: Purpose, evolution, and decay, *Proceedings of ICSE 2019 (International Conference on Software Engineering)*, IEEE, pp. 1211–1221.

[97] He, J., Ghosh, R., Walia, K., Chen, J., Dhadiwal, T., Hazel, A. and Inguva, C. [2024]. Frontiers of large language model-based agentic systems — Construction, efficacy and safety, *Proceedings of CIKM 2024 (International Conference on Information and Knowledge Management)*, ACM, pp. 5526–5529.
URL: *https://doi.org/10.1145/3627673.3679105*

[98] Hebig, R., Ho-Quang, T., Chaudron, M. R. V., Robles, G. and Fernandez, M. A. [2016]. The quest for open source projects that use UML: Mining GitHub, *Proceedings of MODELS 2016 (International Conference on Model Driven Engineering Languages and Systems)*, ACM, pp. 173–183.

[99] Hees, R. V. and Hage, J. [2017]. Stable and predictable Voronoi treemaps for software quality monitoring, *Information and Software Technology* **87**: 242–258.

[100] Herold, S., Blom, M. and Buckley, J. [2016]. Evidence in architecture degradation and consistency checking research: Preliminary results from a literature review, *Proccedings of ECSAW 2016 (European Conference on Software Architecture Workshops)*, ACM, pp. 1–7.

[101] Ho-Quang, T., Chaudron, M. R. V., Samúelsson, I., Hjaltason, J., Karasneh, B. and Osman, H. [2014]. Automatic classification of UML class diagrams from images, *Proceedings of APSEC 2014 (Asia-Pacific Software Engineering Conference)*, Vol. 1, IEEE, pp. 399–406.

[102] Ho-Quang, T., Hebig, R., Robles, G., Chaudron, M. R. V. and Fernandez, M. A. [2017]. Practices and perceptions of UML use in open source projects, *Proceedings of ICSE-SEIP 2017 (International Conference on Software Engineering: Software Engineering in Practice Track)*, IEEE, pp. 203–212.

[103] Hoda, R., Noble, J. and Marshall, S. [2012]. Documentation strategies on agile software development projects, *International Journal of Agile and Extreme Software Development* **1**(1): 23–37.

[104] Hoegl, M. and Gemuenden, H. G. [2001]. Teamwork quality and the success of innovative projects: A theoretical concept and empirical evidence, *Organization Science* **12**(4): 435–449.

[105] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J. and Wang, H. [2024]. Large language models for software engineering: A systematic literature review, *ACM Transactions on Software Engineering and Methodology* **33**(8).
URL: *https://doi.org/10.1145/3695988*

[106] Hruby, P. [1998]. Specification of workflow management systems with UML, *Proceedings of OOPSLA 1998 (Workshop on Implementation and Application of Object-oriented Workflow Management Systems)*, Vol. 2, ACM, pp. 1–11.

[107] Huber, F. and Hagel, G. [2022]. Tool-supported teaching of UML diagrams in software engineering education — A systematic literature review, *Proceedings of the International Convention on Information, Communication and Electronic Technology*, IEEE, pp. 1404–1409.

[108] Jaanu, T., Paasivaara, M. and Lassenius, C. [2012]. Near-synchronicity and distance: Instant messaging as a medium for global software engineering, *Proceedings of GSE 2012 (International Conference on Global Software Engineering)*, IEEE, pp. 149–153.

[109] Jacobson, I. [1992]. *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison Wesley.

[110] Jasser, M. B., Zhen, L. M., Issa, B., Hong, L. M. and Al-Hadi, I. A. A.-Q. [2023]. Quantifying object-oriented system complexity: Introducing a powerful measurement tool, *Proceedings of ICSET 2023 (International Conference on System Engineering and Technology)*, IEEE, pp. 221–226.

[111] Jerding, D. F. and Stasko, J. T. [1994]. Using visualization to foster object-oriented program understanding, *Technical report*, Georgia Institute of Technology.

[112] Jiang, H., Shi, L., Che, M., Zhang, Y. and Wang, Q. [2024]. Bringing open source communication and development together: A cross-platform study on Gitter and GitHub, *IEEE Transactions on Software Engineering* **50**(11): 2807–2826.

[113] Jiang, J. A., Kiene, C., Middler, S., Brubaker, J. R. and Fiesler, C. [2019]. Moderation challenges in voice-based online communities on Discord, *Proceedings of HCI 2019 (Human-Computer Interaction)*, Vol. 3, CSCW, 55, ACM, pp. 1–23.

[114] Jiang, J.-Y., Chen, F., Chen, Y.-Y. and Wang, W. [2018]. Learning to disentangle interleaved conversational threads with a siamese hierarchical network and similarity ranking, *Proceedings of ACL-HLT 2018 (Association for Computational Linguistics: Human Language Technologies)*, pp. 1812–1822.

[115] Jiang, Z. M. and Hassan, A. E. [2006]. Examining the evolution of code comments in PostgreSQL, *Proceedings of MSR 2006 (International Workshop on Mining Software Repositories)*, ACM, pp. 179–180.

[116] Johanssen, J. O., Kleebaum, A., Bruegge, B. and Paech, B. [2017]. Towards the visualization of usage and decision knowledge in continuous software engineering, *Proceedings of VISSOFT 2017 (Working Conference on Software Visualization)*, IEEE, pp. 104–108.

[117] Johnson, B. and Shneiderman, B. [1991]. Tree-maps: A space-filling approach to the visualization of hierarchical information structures, *Proceedings of Vis 1991 (Visualization Conference)*, IEEE Computer Society Press, pp. 284–291.
URL: *https://doi.org/10.1109/VISUAL.1991.175815*

[118] Jolak, R., Savary-Leblanc, M., Dalibor, M., Vincur, J., Hebig, R., Pallec, X. L., Chaudron, M., Gérard, S., Polasek, I. and Wortmann, A. [2022]. The influence of software design

representation on the design communication of teams with diverse personalities, *Proceedings of MODELS 2022 (International Conference on Model Driven Engineering Languages and Systems)*, ACM, pp. 255–265.

[119] Käfer, V., Graziotin, D., Bogicevic, I., Wagner, S. and Ramadani, J. [2018]. Communication in open-source projects — End of the e-mail era?, *Proceedings of ICSE 2018 Companion (International Conference on Software Engineering Companion)*, ACM, pp. 242–243.

[120] Kajko-Mattsson, M. [2005]. A survey of documentation practice within corrective maintenance, *Empirical Software Engineering* **10**(1): 31–55.

[121] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M. and Damian, D. [2014]. The promises and perils of mining GitHub, *Proceedings of MSR 2014 (Working Conference on Mining Software Repositories)*, ACM, pp. 92–101.

[122] Kalnins, A. and Vitolins, V. [2006]. Use of UML and model transformations for workflow process definitions, *arXiv preprint* .

[123] Kernighan, B. W. and Plauger, P. J. [1974]. Programming style, *Proceedings of SIGCSE 1974 (Technical Symposium on Computer Science Education)*, ACM, pp. 90–96.

[124] Khan, J. Y. and Uddin, G. [2023]. Automatic code documentation generation using GPT-3, *Proceedings of ASE 2022 (International Conference on Automated Software Engineering)*, ACM.
URL: *https://doi.org/10.1145/3551349.3559548*

[125] Klotins, E., Gorschek, T., Sundelin, K. and Falk, E. [2022]. Towards cost-benefit evaluation for continuous software engineering activities, *Empirical Software Engineering* **27**(157): 1–40.

[126] Knight, C. [1998]. Visualisation for program comprehension: Information and issues, *Technical report*, Centre for Software Maintenance, University of Durham.

[127] Kouters, E., Vasilescu, B., Serebrenik, A. and Van Den Brand, M. G. J. [2012]. Who's who in Gnome: Using LSA to merge software repository identities, *Proceedings of ICSM 2012 (International Conference on Software Maintenance)*, IEEE, pp. 592–595.

[128] Kraemer, E. and Stasko, J. T. [1994]. Issues in visualization for the comprehension of parallel programs, *Proceedings of WPC 1994 (Workshop on Program Comprehension)*, IEEE, pp. 116–125.

[129] Kummerfeld, J. K., Gouravajhala, S. R., Peper, J. J., Athreya, V., Gunasekara, C., Ganhotra, J., Patel, S. S., Polymenakos, L. C. and Lasecki, W. [2019]. A large-scale corpus for conversation disentanglement, *Proceedings of ACL 2019 (Annual Meeting of the Association for Computational Linguistics)*, ACL, pp. 3846–3856.

[130] Lange, C. F. J., Chaudron, M. R. V. and Muskens, J. [2006]. In practice: UML software architecture and design description, *IEEE Software* **23**(2): 40–46.

[131] Lanza, M. [2004]. CodeCrawler — Polymetric views in action, *Proceedings of ASE 2004 (International Conference on Automated Software Engineering)*, IEEE, pp. 394–395.

[132] Lanza, M. and Ducasse, S. [2003]. Polymetric views — A lightweight visual approach to reverse engineering, *IEEE Transactions on Software Engineering* **29**(9): 782–795.

[133] Ledgard, H. F. [1975]. *Programming Proverbs*, Hayden Books.

[134] Leinonen, J., Denny, P., MacNeil, S., Sarsa, S., Bernstein, S., Kim, J., Tran, A. and Hellas, A. [2023]. Comparing code explanations created by students and large language models, *Proceedings of ITiCSE 2023 (Conference on Innovation and Technology in Computer Science Education)*, ACM, pp. 124–130.
URL: *https://doi.org/10.1145/3587102.3588785*

[135] Lethbridge, T. C., Singer, J. and Forward, A. [2003]. How software engineers use documentation: The state of the practice, *IEEE Software* **20**(6): 35–39.

[136] Leuf, B. and Cunningham, W. [2001]. *The Wiki Way: Quick Collaboration on the Web*, Addison-Wesley Longman Publishing Co., Inc.

[137] Li, R., Liang, P., Soliman, M. and Avgeriou, P. [2021]. Understanding architecture erosion: The practitioners' perceptive, *Proceedings of ICPC 2021 (International Conference on Program Comprehension)*, IEEE, pp. 311–322.

[138] Li, R., Liang, P., Soliman, M. and Avgeriou, P. [2022]. Understanding software architecture erosion: A systematic mapping study, *Journal of Software: Evolution and Process* **34**(3): e2423.

[139] Li, R., Soliman, M., Liang, P. and Avgeriou, P. [2022]. Symptoms of architecture erosion in code reviews: A study of two OpenStack projects, *Proceedings of ICSA 2022 (International Conference on Software Architecture)*, IEEE, pp. 24–35.

[140] Li, T., Gu, J.-C., Zhu, X., Liu, Q., Ling, Z.-H., Su, Z. and Wei, S. [2021]. DialBERT: A hierarchical pre-trained model for conversation disentanglement.
URL: *https://arxiv.org/abs/2004.03760*

[141] Lill, A., Meyer, A. N. and Fritz, T. [2024]. On the helpfulness of answering developer questions on Discord with similar conversations and posts from the past, *Proceedings of ICSE 2024 (International Conference on Software Engineering)*, ACM.

[142] Lima, M., Steinmacher, I., Ford, D., Liu, E., Vorreuter, G., Conte, T. and Gadelha, B. [2022]. Looking for related discussions on GitHub Discussions, *arXiv*.

[143] Lima, M., Steinmacher, I., Ford, D., Vorreuter, G., Gonçalves, L., Conte, T. and Gadelha, B. [2025]. How are discussions linked? A link analysis study on GitHub Discussions, *Journal of Systems and Software* **219**: 112196.

[144] Lin, B., Zagalsky, A., Storey, M.-A. and Serebrenik, A. [2016]. Why developers are slacking off: Understanding how software teams use Slack, *Proceedings of CSCW/SCC 2016 (Conference on Computer Supported Cooperative Work and Social Computing Companion)*, ACM, pp. 333–336.

[145] Lin, B., Zampetti, F., Bavota, G., Di Penta, M. and Lanza, M. [2019]. Pattern-based mining of opinions in Q&A websites, *Proceedings of ICSE 2019 (International Conference on Software Engineering)*, IEEE, pp. 548–559.

[146] Lin, F., Kim, D. J., Tse-Husn and Chen [2024]. SOEN-101: Code generation by emulating software process models using large language model agents, arxiv:2403.15852.
URL: *https://arxiv.org/abs/2403.15852*

[147] Lindsjørn, Y., Sjøberg, D. I. K., Dingsøyr, T., Bergersen, G. R. and Dybå, T. [2016]. Teamwork quality and project success in software development: A survey of agile development teams, *Journal of Systems and Software* **122**: 274–286.

[148] Linos, P., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P. and Tulula, P. [1993]. Facilitating the comprehension of C-programs: An experimental study, *Proceedings of ICPC 1993 (Workshop on Program Comprehension)*, IEEE, pp. 55–63.

[149] Liu, H., Shi, Z., Gu, J.-C., Liu, Q., Wei, S. and Zhu, X. [2020]. End-to-end transition-based online dialogue disentanglement, *Proceedings of IJCAI 2021 (International Joint Conference on Artificial Intelligence)*, ACM, pp. 3868–3874.

[150] Liu, H., Shi, Z. and Zhu, X. [2021]. Unsupervised conversation disentanglement through co-training, *Proceedings of EMNLP 2021 (Conference on Empirical Methods in Natural Language Processing)*, ACL, pp. 2345–2356.

[151] Luo, Q., Ye, Y., Liang, S., Zhang, Z., Qin, Y., Lu, Y., Wu, Y., Cong, X., Lin, Y., Zhang, Y., Che, X., Liu, Z. and Sun, M. [2024]. RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation, arxiv:2402.16667.
URL: *https://arxiv.org/abs/2402.16667*

[152] MacNeil, S., Tran, A., Mogil, D., Bernstein, S., Ross, E. and Huang, Z. [2022]. Generating diverse code explanations using the GPT-3 large language model, *Proceedings of ICER 2022 (Conference on International Computing Education Research)*, ACM, pp. 37–39.
URL: *https://doi.org/10.1145/3501709.3544280*

[153] Manes, S. S. and Baysal, O. [2019]. How often and what StackOverflow posts do developers reference in their GitHub projects?, *Proceedings of MSR 2019 (International Conference on Mining Software Repositories)*, IEEE, pp. 235–239.

[154] McLuhan, M. [1964]. *Understanding Media*, Gingko Press.

[155] Mehri, S. and Carenini, G. [2017]. Chat disentanglement: Identifying semantic reply relationships with random forests and recurrent neural networks, *Proceedings of JCNLP 2017 (International Joint Conference on Natural Language Processing)*, ACL, pp. 615–623.

[156] Meldrum, S., Licorish, S. A., Owen, C. A. and Savarimuthu, B. T. R. [2020]. Understanding Stack Overflow code quality: A recommendation of caution, *Science of Computer Programming* **199**: 102516.

[157] Menzies, R. and Zarb, M. [2020]. Professional communication tools in higher education: A case study in implementing Slack in the curriculum, *Proceedings of FIE 2020 (Frontiers in Education Conference)*, IEEE, pp. 1–8.

[158] Mezouar, M. E., Zhang, F. and Zou, Y. [2018]. Are tweets useful in the bug fixing process? An empirical study on Firefox and Chrome, *Empirical Software Engineering* **23**(3): 1704–1742.

[159] Moreira, T. G., Wehrmeister, M. A., Pereira, C. E., Petin, J.-F. and Levrat, E. [2010]. Automatic code generation for embedded systems: From UML specifications to VHDL code, *Proceedings of INDIN 2010 (International Conference on Industrial Informatics)*, IEEE, pp. 1085–1090.

[160] Moslehi, P., Adams, B. and Rilling, J. [2016]. On mining crowd-based speech documentation, *Proceedings of MSR 2016 (Working Conference on Mining Software Repositories)*, ACM, pp. 259–268.

[161] Mutton, P. [2004]. Inferring and visualizing social networks on internet relay chat, *Proceedings of IV 2004 (International Conference on Information Visualisation)*, IEEE, pp. 35–43.

[162] Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B. and Myers, B. [2024]. Using an LLM to help with code understanding, *Proceedings of ICSE 2024 (International Conference on Software Engineering)*, ACM.
URL: *https://doi.org/10.1145/3597503.3639187*

[163] Naur, P. and Randell, B. [1968]. Software engineering, report on a conference sponsored by the NATO Science Committee, Garmisch, Germany.

[164] Neu, S., Lanza, M., Hattori, L. and D'Ambros, M. [2011]. Telling stories about GNOME with Complicity, *Proceedings of VISSOFT 2011 (International Workshop on Visualizing Software for Understanding and Analysis)*, IEEE, pp. 1–8.

[165] Nugroho, Y. S., Islam, S., Nakasai, K., Rehman, I., Hata, H., Kula, R. G., Nagappan, M. and Matsumoto, K. [2021]. How are project-specific forums utilized? A study of participation, content, and sentiment in the Eclipse ecosystem, *Empirical Software Engineering* **26**(6): 132.

[166] Nurmuliani, N., Zowghi, D. and Williams, S. P. [2004]. Using card sorting technique to classify requirements change, *Proceedings of IREC 2004 (International Requirements Engineering Conference)*, IEEE, pp. 240–248.

[167] Opler, A. [1968]. Acceptance testing of large programming systems, *Software Engineering, Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany*, pp. 209–211.

[168] Ozkaya, M. [2019]. Are the UML modelling tools powerful enough for practitioners? A literature review, *IET Software* **13**(5): 338–354.

[169] Ozkaya, M. and Erata, F. [2020]. A survey on the practical use of UML for different software architecture viewpoints, *Information and Software Technology* **121**: 106275.

[170] Pagano, D. and Maalej, W. [2011]. How do developers blog? An exploratory study, *Proceedings of MSR 2011 (Working Conference on Mining Software Repositories)*, ACM, pp. 123–132.

[171] Parnas, D. L. [2009]. Document based rational software development, *Knowledge-Based Systems* **22**(3): 132–141.

[172] Parnas, D. L. and Madey, J. [1995]. Functional documents for computer systems, *Science of Computer Programming* **25**(1): 41–61.

[173] Parnin, C., Treude, C., Grammel, L. and Storey, M.-A. [2012]. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow, *Technical report*, Georgia Institute of Technology.

[174] Parra, E., Alahmadi, M., Ellis, A. and Haiduc, S. [2022]. A comparative study and analysis of developer communications on Slack and Gitter, *Empirical Software Engineering* **27**(2): 1–33.

[175] Parra, E., Ellis, A. and Haiduc, S. [2020]. GitterCom: A dataset of open source developer communications in Gitter, *Proceedings of MSR 2020 (International Conference on Mining Software Repositories)*, ACM, pp. 563–567.

[176] Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R. and Lanza, M. [2014]. Mining StackOverflow to turn the IDE into a self-confident programming prompter, *Proceedings of MSR 2014 (Working Conference on Mining Software Repositories)*, ACM, pp. 102–111.

[177] Poo-Caamaño, G., Knauss, E., Singer, L. and German, D. M. [2017]. Herding cats in a FOSS ecosystem: A tale of communication and coordination for release management, *Journal of Internet Services and Applications* **8**(1): 1–24.

[178] Poo-Caamaño, G., Singer, L., Knauss, E. and German, D. M. [2016]. Herding cats: A case study of release management in an open collaboration ecosystem, *Open Source Systems: Integrating Communities*, Springer, pp. 147–162.

[179] Prause, C. R. and Durdik, Z. [2012]. Architectural design and documentation: Waste in agile development?, *Proceedings of ICSSP 2012 (International Conference on Software and System Process)*, IEEE, pp. 130–134.

[180] Raglianti, M. [2022]. Topology of the Documentation Landscape, *Proceedings of ICSE 2022 Companion (International Conference on Software Engineering Companion)*, ACM, pp. 297–299.

[181] Raglianti, M., Minelli, R., Nagy, C. and Lanza, M. [2021]. Visualizing Discord servers, *Proceedings of VISSOFT 2021 (Working Conference on Software Visualization)*, IEEE, pp. 150–154.

[182] Raglianti, M., Nagy, C., Minelli, R. and Lanza, M. [2022a]. DiscOrDance: Visualizing software developers communities on Discord, *Proceedings of ICSME 2022 (International Conference on Software Maintenance and Evolution)*, IEEE, pp. 474–478.

[183] Raglianti, M., Nagy, C., Minelli, R. and Lanza, M. [2022b]. Using Discord conversations as program comprehension aid, *Proceedings of ICPC 2022 (International Conference on Program Comprehension)*, ACM, pp. 597–601.

[184] Raglianti, M., Nagy, C., Minelli, R., Lin, B. and Lanza, M. [2023]. On the rise of modern software documentation, *Proceedings of ECOOP 2023 (European Conference on Object-Oriented Programming)*, Dagstuhl Publishing, pp. 43:1–43:24.

[185] Raibulet, C., Fontana, F. A. and Zanoni, M. [2017]. Model-driven reverse engineering approaches: A systematic literature review, *IEEE Access* **5**: 14516–14542.

[186] Ramsey, H. R., Atwood, M. E. and Van Doren, J. R. [1978]. A comparative study of flowcharts and program design languages for the detailed procedural specification of computer programs, *Technical report*, Science Applications Inc. Englewood Co.

[187] Raymond, E. [1999]. The cathedral and the bazaar, *Knowledge, Technology & Policy* **12**(3): 23–49.

[188] Reifer, D. J., Basili, V. R., Boehm, B. W. and Clark, B. [2003]. Eight lessons learned during COTS-based systems maintenance, *IEEE Software* **20**(5): 94–96.

[189] Riggio, E., Raglianti, M. and Lanza, M. [2023]. Conversation disentanglement as-a-service, *Proceedings of ICPC 2023 (International Conference on Program Comprehension)*, IEEE, pp. 59–63.

[190] Robert, L. P. and Dennis, A. R. [2005]. Paradox of richness: A cognitive model of media choice, *IEEE Transactions on Professional Communication* **48**(1): 10–21.

[191] Robillard, M. P. [2009]. What makes APIs hard to learn? Answers from developers, *IEEE Software* **26**(6): 27–34.

[192] Robillard, M. P. and DeLine, R. [2011]. A field study of API learning obstacles, *Empirical Software Engineering* **16**(6): 703–732.

[193] Robillard, M. P., Marcus, A., Treude, C., Bavota, G., Chaparro, O., Ernst, N., Gerosa, M. A., Godfrey, M., Lanza, M., Linares-Vásquez, M., Murphy, G. C., Moreno, L., Shepherd, D. and Wong, E. [2017]. On-demand developer documentation, *Proceedings of ICSME 2017 (International Conference on Software Maintenance and Evolution)*, IEEE, pp. 479–483.

[194] Robles, G., Chaudron, M. R. V., Jolak, R. and Hebig, R. [2023]. A reflection on the impact of model mining from GitHub, *Information and Software Technology* **164**: 107317.

[195] Robles, G., Ho-Quang, T., Hebig, R., Chaudron, M. R. V. and Fernandez, M. A. [2017]. An extensive dataset of UML models in GitHub, *Proceedings of MSR 2017 (International Conference on Mining Software Repositories)*, IEEE, pp. 519–522.

[196] Rodolfo Masera, T. [2023]. *Mapping the documentation landscape of open source projects*, Master's thesis, Master of Science in Software and Data Engineering, Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland.
URL: *https://www.inf.usi.ch/faculty/lanza/Downloads/MSc/Mase2023a.pdf*

[197] Romeo, J. [2023]. *On the usage of uml diagrams in open source projects*, Master's thesis, Master of Science in Software and Data Engineering, Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland.
URL: *https://www.inf.usi.ch/faculty/lanza/Downloads/MSc/Rome2023a.pdf*

[198] Romeo, J., Raglianti, M., Nagy, C. and Lanza, M. [2024]. Capturing and understanding the drift between design, implementation, and documentation, *Proceedings of ICPC 2024 (International Conference on Program Comprehension)*, ACM, pp. 382–386.

[199] Romeo, J., Raglianti, M., Nagy, C. and Lanza, M. [2025]. Investigating the past, present, and future of uml documentation in open source software, *Proceedings of ICSE 2025 (International Conference on Software Engineering)*, IEEE, p. in press.

[200] Rosik, J., Le Gear, A., Buckley, J., Babar, M. A. and Connolly, D. [2011]. Assessing architectural drift in commercial software development: A case study, *Software: Practice and Experience* **41**(1): 63–86.

[201] Ross, S. I., Martinez, F., Houde, S., Muller, M. and Weisz, J. D. [2023]. The programmer's assistant: Conversational interaction with a large language model for software development, *Proceedings of IUI 2023 (International Conference on Intelligent User Interfaces)*, ACM, pp. 491–514.
URL: *https://doi.org/10.1145/3581641.3584037*

[202] Royce, W. W. [1970]. Managing the development of large software systems: Concepts and techniques, *Proceedings of WESCON 1970 (Western Electronic Show and Convention)*, IEEE, pp. 328–338.

[203] Rubin, E. and Rubin, H. [2011]. Supporting agile software development through active documentation, *Requirements Engineering* **16**(2): 117–132.

[204] Rugaber, S. [1992]. Program comprehension for reverse engineering, *Proceedings of AAAI 1992 (Workshop on AI and Automated Program Understanding)*, Citeseer, pp. 106–110.

[205] Rugaber, S. [1995]. Program comprehension, *Encyclopedia of Computer Science and Technology* **35**(20): 341–368.

[206] Rukmono, S. A. and Chaudron, M. R. V. [2022]. Guiding peer-feedback in learning software design using UML, *Proceedings of ICSE-SEET 2022 (International Conference on Software Engineering: Software Engineering Education and Training)*, ACM, pp. 122–133.

[207] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. E. [1991]. *Object-Oriented Modeling and Design*, Prentice-Hall.

[208] Sabir, U., Azam, F., Haq, S. U., Anwar, M. W., Butt, W. H. and Amjad, A. [2019]. A model driven reverse engineering framework for generating high level UML models from Java source code, *IEEE Access* **7**: 158931–158950.

[209] Sahar, H., Hindle, A. and Bezemer, C.-P. [2021]. How are issue reports discussed in Gitter chat rooms?, *Journal of Systems and Software* **172**: 110852.

[210] Sajadi, A., Damevski, K. and Chatterjee, P. [2023]. Towards understanding emotions in informal developer interactions: A Gitter chat study, *Proceedings of ESEC/FSE 2023 (Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering)*, ACM, pp. 2097–2101.

[211] Sajedi Badashian, A., Hindle, A. and Stroulia, E. [2016]. Crowdsourced bug triaging: Leveraging Q&A platforms for bug assignment, *Proceedings of FASE 2016 (International Conference on Fundamental Approaches to Software Engineering)*, Springer, pp. 231–248.

[212] Saunders, B., Sim, J., Kingstone, T., Baker, S., Waterfield, J., Bartlam, B., Burroughs, H. and Jinks, C. [2018]. Saturation in qualitative research: Exploring its conceptualization and operationalization, *Quality & Quantity* **52**(4): 1893–1907.

[213] Scanniello, G., Gravino, C., Genero, M., Cruz-Lemus, J. A. and Tortora, G. [2014]. On the impact of UML analysis models on source-code comprehensibility and modifiability, *ACM Transactions on Software Engineering and Methodology* **23**(2): 1–26.

[214] Scanniello, G., Gravino, C., Genero, M., Cruz-Lemus, J. A., Tortora, G., Risi, M. and Dodero, G. [2018]. Do software models based on the UML aid in source-code comprehensibility? Aggregating evidence from 12 controlled experiments, *Empirical Software Engineering* **23**(5): 2695–2733.

[215] Scheibel, W., Trapp, M., Limberger, D. and Döllner, J. [2020]. A taxonomy of treemap visualization techniques, *Proceedings of VISIGRAPP 2020 (International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications)*, Vol. 3: IVAPP, pp. 273–280.

[216] Scheibel, W., Weyand, C. and Döllner, J. [2018]. EvoCells — A treemap layout algorithm for evolving tree data, *Proceedings of VISIGRAPP 2018 (International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications)*, Vol. 3: IVAPP, SCITEPRESS, pp. 273–280.

[217] Schwaber, K. [2004]. *Agile Project Management with SCRUM*, Microsoft Press.

[218] Selic, B. [2009]. Agile documentation, anyone?, *IEEE Software* **26**(6): 11–12.

[219] Selig, F. [1968]. Documentation standards, *Software Engineering, Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany*, pp. 209–211.

[220] Sharma, A., Tian, Y. and Lo, D. [2015]. What's hot in software engineering Twitter space?, *Proceedings of ICSME 2015 (International Conference on Software Maintenance and Evolution)*, IEEE, pp. 541–545.

[221] Shaw, M. [1996]. Three patterns that help explain the development of software engineering, *History of Software Engineering* pp. 52–56.

[222] Shen, D., Yang, Q., Sun, J.-T. and Chen, Z. [2006]. Thread detection in dynamic text message streams, *Proceedings of SIGIR 2006 (International Conference on Research and Development in Information Retrieval)*, ACM, pp. 35–42.

[223] Sheppard, S. B., Kruesi, E. and Bailey, J. W. [1982]. An empirical evaluation of software documentation formats, *Proceedings of CHI 1982 (Conference on Human Factors in Computing Systems)*, ACM, pp. 121–124.

[224] Sherertz, D. D. [1980]. The evolution of a language standard: MUMPS in the 1980s, *Proceedings of the ACM 1980 Annual Conference*, ACM, pp. 101–104.

[225] Shi, L., Chen, X., Yang, Y., Jiang, H., Jiang, Z., Niu, N. and Wang, Q. [2021]. A first look at developers' live chat on Gitter, *Proceedings of ESEC/FSE 2021 (European Software Engineering Conference and Symposium on the Foundations of Software Engineering)*, ACM, pp. 391–403.

[226] Shihab, E., Jiang, Z. M. and Hassan, A. E. [2009a]. On the use of Internet Relay Chat (IRC) meetings by developers of the GNOME GTK+ project, *Proceedings of MSR 2009 (International Working Conference on Mining Software Repositories)*, IEEE, pp. 107–110.

[227] Shihab, E., Jiang, Z. M. and Hassan, A. E. [2009b]. Studying the use of developer IRC meetings in open source projects, *Proceedings of ICSM 2009 (International Conference on Software Maintenance)*, IEEE, pp. 147–156.

[228] Shneiderman, B. [1992]. Tree visualization with tree-maps: 2-D space-filling approach, *ACM Transactions on Graphics* **11**(1): 92–99.

[229] Shneiderman, B., Mayer, R., McKay, D. and Heller, P. [1977]. Experimental investigations of the utility of detailed flowcharts in programming, *Communications of the ACM* **20**(6): 373–381.

[230] Smith, C. P. [1980]. A software science analysis of programming size, *Proceedings of ACM-AC 1980 (ACM Annual Conference)*, ACM, pp. 179–185.

[231] Sommerville, I. [2015]. *Software Engineering*, 10th edn, Pearson.

[232] Spencer, D. [2009]. *Card Sorting: Designing Usable Categories*, Rosenfeld Media.

[233] Steinbeck, M. and Koschke, R. [2021]. Javadoc violations and their evolution in open-source software, *Proceedings of SANER 2021 (International Conference on Software Analysis, Evolution and Reengineering)*, IEEE, pp. 249–259.

[234] Stephany, F., Mens, T. and Gîrba, T. [2009]. Maispion: A tool for analysing and visualising open source software developer communities, *Proceedings of IWST 2009 (International Workshop on Smalltalk Technologies)*, ACM, pp. 50–57.

[235] Stettina, C. J. and Heijstek, W. [2011]. Necessary and neglected? An empirical study of internal documentation in agile software development teams, *Proceedings of SIGDOC 2011 (International Conference on Design of Communication)*, ACM, pp. 159–166.

[236] Storey, M.-A. D. and Muller, H. A. [1995]. Manipulating and documenting software structures using SHriMP views, *Proceedings of ICSM 1995 (International Conference on Software Maintenance)*, IEEE, pp. 275–284.

[237] Storey, M.-A. D., Wong, K., Fracchia, F. D. and Muller, H. A. [1997]. On integrating visualization techniques for effective software exploration, *Proceedings of VIZ 1997 (Visualization Conference, Information Visualization Symposium and Parallel Rendering Symposium)*, IEEE, pp. 38–45.

[238] Storey, M.-A., Fracchia, D. F. and Müller, H. A. [1999]. Cognitive design elements to support the construction of a mental model during software exploration, *Journal of Systems and Software* **44**(3): 171–185.

[239] Storey, M.-A., Singer, L., Cleary, B., Figueira Filho, F. and Zagalsky, A. [2014]. The (R)evolution of social media in software engineering, *Proceedings of FOSE 2014 (Future of Software Engineering Proceedings)*, ACM, pp. 100–116.

[240] Storey, M.-A., Zagalsky, A., Filho, F. F., Singer, L. and German, D. M. [2017]. How social and communication channels shape and challenge a participatory culture in software development, *IEEE Transactions on Software Engineering* **43**(2): 185–204.

[241] Stray, V. and Moe, N. B. [2020]. Understanding coordination in global software engineering: A mixed-methods study on the use of meetings and Slack, *Journal of Systems and Software* **170**: 110717.

[242] Stray, V., Moe, N. B. and Noroozi, M. [2019]. Slack me if you can! Using enterprise social networking tools in virtual agile teams, *Proceedings of ICGSE 2019 (International Conference on Global Software Engineering)*, IEEE, pp. 111–121.

[243] Su, Y., Wan, C., Sethi, U., Lu, S., Musuvathi, M. and Nath, S. [2023]. HotGPT: How to make software documentation more useful with a large language model?, *Proceedings of HOTOS 2023 (Workshop on Hot Topics in Operating Systems)*, ACM, pp. 87–93.
URL: *https://doi.org/10.1145/3593856.3595910*

[244] Subash, K. M., Kumar, L. P., Vadlamani, S. L., Chatterjee, P. and Baysal, O. [2022]. DISCO: A dataset of Discord chat conversations for software engineering research, *Proceedings of MSR 2022 (International Conference on Mining Software Repositories)*, IEEE, pp. 227–231.

[245] Systa, T., Yu, P. and Muller, H. [2000]. Analyzing Java software by combining metrics and program visualization, *Proceedings of ECSMR 2000 (European Conference on Software Maintenance and Reengineering)*, IEEE, pp. 199–208.

[246] Talebirad, Y. and Nadiri, A. [2023]. Multi-agent collaboration: Harnessing the power of intelligent LLM agents, arxiv:2306.03314.
URL: *https://arxiv.org/abs/2306.03314*

[247] Tantisuwankul, J., Nugroho, Y. S., Kula, R. G., Hata, H., Rungsawang, A., Leelaprute, P. and Matsumoto, K. [2019]. A topological analysis of communication channels for knowledge sharing in contemporary GitHub projects, *Journal of Systems and Software* **158**: 110416.

[248] Tausworthe, R. C. [1976]. Standard classification of software documentation, *Technical report*, NASA.

[249] Teixeira, J. A. and Karsten, H. [2019]. Managing to release early, often and on time in the OpenStack software ecosystem, *Journal of Internet Services and Applications* **10**(1): 7.

[250] Thelwall, M. and Vaughan, L. [2004]. A fair history of the web? Examining country balance in the Internet Archive, *Library & Information Science Research* **26**(2): 162–176.

[251] Tian, Y., Achananuparp, P., Lubis, I. N., Lo, D. and Lim, E.-P. [2012]. What does software engineering community microblog about?, *Proceedings of MSR 2012 (Working Conference on Mining Software Repositories)*, IEEE, pp. 247–250.

[252] Tran, K.-T., Dao, D., Nguyen, M.-D., Pham, Q.-V., O'Sullivan, B. and Nguyen, H. D. [2025]. Multi-agent collaboration mechanisms: A survey of llms, arxiv:2501.06322.
URL: *https://arxiv.org/abs/2501.06322*

[253] Treude, C., Robillard, M. P. and Dagenais, B. [2015]. Extracting development tasks to navigate software documentation, *IEEE Transactions on Software Engineering* **41**(6): 565–581.

[254] Treude, C. and Storey, M.-A. [2011]. Effective communication of software development knowledge through community portals, *Proceedings of ESEC/FSE 2011 (European Software Engineering Conference and Symposium on the Foundations of Software Engineering)*, ACM, pp. 91–101.

[255] Tryggeseth, E. [1997]. Report from an experiment: Impact of documentation on maintenance, *Empirical Software Engineering* **2**(2): 201–207.

[256] Tua, D. P., Minelli, R. and Lanza, M. [2021]. Voronoi evolving treemaps, *Proceedings of VISSOFT 2021 (Working Conference on Software Visualization)*, IEEE, pp. 1–5.

[257] Tufte, E. [1990]. *Envisioning Information*, Graphics Press.

[258] Uddin, G. and Robillard, M. P. [2015]. How API documentation fails, *IEEE Software* **32**(4): 68–75.

[259] Watson, A. [2008]. Visual modelling: Past, present and future, *White Paper UML Object Modeling Group* pp. 1–6.
URL: *https://www.omg.org/UML/Visual_Modeling.pdf*

[260] Wilson, J. T. [1965]. A new class of faults and their bearing on continental drift, *Nature* **207**: 343–347.

[261] Wood, J. R. and Wood, L. E. [2008]. Card sorting: Current practices and beyond, *Journal of Usability Studies* **4**(1): 1–6.

[262] Yang, Z., Wang, C., Shi, J., Hoang, T., Kochhar, P., Lu, Q., Xing, Z. and Lo, D. [2023]. What do users ask in open-source AI repositories? An empirical study of GitHub Issues, *arXiv preprint* .
URL: *https://arxiv.org/abs/2303.09795*

[263] Yu, L., Ramaswamy, S., Mishra, A. and Mishra, D. [2011]. Communications in global software development: An empirical study using GTK+ OSS repository, *Proceedings of OTM 2011 (On the Move to Meaningful Internet Systems)*, Springer, pp. 218–227.

[264] Zagalsky, A., German, D. M., Storey, M.-A., Teshima, C. G. and Poo-Caamaño, G. [2018]. How the R community creates and curates knowledge: An extended study of Stack Overflow and mailing lists, *Empirical Software Engineering* **23**(2): 953–986.

[265] Zhi, J., Garousi-Yusifoğlu, V., Sun, B., Garousi, G., Shahnewaz, S. and Ruhe, G. [2015]. Cost, benefits and quality of software development documentation: A systematic mapping, *Journal of Systems and Software* **99**: 175–198.

[266] Zhou, S., Vasilescu, B. and Kästner, C. [2020]. How has forking changed in the last 20 years? A study of hard forks on github, *Proceedings of ICSE 2020 (International Conference on Software Engineering)*, ACM, pp. 445–456.

[267] Zhu, H., Nan, F., Wang, Z., Nallapati, R. and Xiang, B. [2020]. Who did they respond to? Conversation structure modeling using masked hierarchical transformer, *Proceedings of AAAI 2020 (Conference on Artificial Intelligence)*, Vol. 34(05), PKP|PS, pp. 9741–9748.

[268] Zhu, R., Lau, J. H. and Qi, J. [2021]. Findings on conversation disentanglement, arXiv:2112.05346.
URL: *https://arxiv.org/abs/2112.05346*

[269] Zimmerle, C., Gama, K., Castor, F. and Filho, J. M. M. [2022]. Mining the usage of reactive programming APIs: A study on GitHub and Stack Overflow, *Proceedings of MSR 2022 (International Conference on Mining Software Repositories)*, ACM, pp. 203–214.

URL References

[270] Scott Chacon and Ben Straub, "Git Book — What is Git?" URL: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git

[271] Wikipedia, "Blog." URL: https://en.wikipedia.org/wiki/Blog

[272] ——, "Medium." URL: https://en.wikipedia.org/wiki/Medium_(website)

[273] ——, "WordPress." URL: https://en.wikipedia.org/wiki/WordPress

[274] ——, "Discord." URL: https://en.wikipedia.org/wiki/Discord

[275] ——, "Gitter." URL: https://en.wikipedia.org/wiki/Gitter

[276] ——, "Internet Relay Chat (IRC)." URL: https://en.wikipedia.org/wiki/Internet_Relay_Chat

[277] ——, "Telegram." URL: https://en.wikipedia.org/wiki/Telegram_(software)

[278] ——, "Imgur." URL: https://en.wikipedia.org/wiki/Imgur

[279] ——, "Vimeo." URL: https://en.wikipedia.org/wiki/Vimeo

[280] ——, "YouTube." URL: https://en.wikipedia.org/wiki/YouTube

[281] ——, "Facebook." URL: https://en.wikipedia.org/wiki/Facebook

[282] ——, "Instagram." URL: https://en.wikipedia.org/wiki/Instagram

[283] ——, "TikTok." URL: https://en.wikipedia.org/wiki/TikTok

[284] ——, "Twitter." URL: https://en.wikipedia.org/wiki/Twitter

[285] phpBB Limited, "PHP Bulletin Board." URL: https://www.phpbb.com

[286] GitHub, "GitHub Discussions." URL: https://docs.github.com/en/discussions

[287] Wikipedia, "Google Groups." URL: https://en.wikipedia.org/wiki/Google_Groups

[288] ——, "BugZilla." URL: https://en.wikipedia.org/wiki/Bugzilla

[289] ——, "Jira." URL: https://en.wikipedia.org/wiki/Jira_(software)

[290] GitHub, "GitHub Pull Requests Model." URL: https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests

[291] Wikipedia, "Bitbucket." URL: https://en.wikipedia.org/wiki/Bitbucket

[292] ——, "GitHub." URL: https://en.wikipedia.org/wiki/GitHub

[293] ——, "Launchpad." URL: https://en.wikipedia.org/wiki/Launchpad_(website)

[294] ——, "SourceForge." URL: https://en.wikipedia.org/wiki/SourceForge

[295] ——, "Weblate." URL: https://en.wikipedia.org/wiki/Weblate

[296] ——, "Wikipedia Vandalism." URL: https://en.wikipedia.org/wiki/Vandalism_on_Wikipedia

[297] ——, "Wikipedia." URL: https://en.wikipedia.org/wiki/Wikipedia

[298] ——, "Data Binning." URL: https://en.wikipedia.org/wiki/Data_binning

[299] GitHub, "Content for the Demisto Platform." URL: https://github.com/demisto/content

[300] ——, "LeetCode Solutions." URL: https://github.com/doocs/leetcode

[301] ——, "Quantum Mechanical Keyboard Firmware." URL: https://github.com/qmk/qmk_firmware

[302] ——, "Fish Shell." URL: https://github.com/fish-shell/fish-shell/

[303] Wikipedia, "GNU Compiler Collection (GCC)." URL: https://en.wikipedia.org/wiki/GNU_Compiler_Collection

[304] freeCodeCamp, "Our experience with Slack." URL: https://www.freecodecamp.org/news/so-yeah-we-tried-slack-and-we-deeply-regretted-it-391bcc714c81

[305] Papyrs, "Easy company intranet & internal team wiki for Slack." URL: https://papyrs.com/slack-wiki-intranet/

[306] T. Abbott, "Why Slack's free plan change is causing an exodus." URL: https://blog.zulip.com/2022/08/26/why-slacks-free-plan-change-is-causing-an-exodus/

[307] D. Curry, "Slack revenue and usage statistics (2022)." URL: https://www.businessofapps.com/data/slack-statistics/

[308] B. Lovin, "Join us on our new journey." URL: https://web.archive.org/web/20220927203327/https://spectrum.chat/spectrum/general/join-us-on-our-new-journey~e4ca0386-f15c-4ba8-8184-21cf5fa39cf5

[309] GitHub, Inc., "Fork a repo." URL: https://docs.github.com/en/get-started/quickstart/fork-a-repo

[310] Zyte, "Scrapy." URL: https://scrapy.org

[311] PyGithub, "PyGithub." URL: https://github.com/PyGithub/PyGithub

[312] Discord, "Invites 101." URL: https://support.discord.com/hc/en-us/articles/208866998-Invites-101

[313] M. Raglianti, C. Nagy, R. Minelli, B. Lin, and M. Lanza, "Replication package." URL: https://figshare.com/s/33c8af534dba61d72c41

[314] Python Software Foundation, "Python Package Index." URL: https://pypi.org/

[315] Sonatype, “Maven Central Repository.” URL: https://central.sonatype.dev/

[316] Travis CI, “Travis CI.” URL: https://www.travis-ci.com/

[317] Codecov, “Codecov.” URL: https://about.codecov.io/

[318] Slack Technologies, “Slack.” URL: https://slack.com/

[319] Discord, Inc., “Discord.” URL: https://discord.com/

[320] A Medium Corporation, “Medium.” URL: https://medium.com/

[321] New Vector, Ltd., “Gitter.” URL: https://gitter.im/

[322] Meta, “Facebook.” URL: https://www.facebook.com/

[323] GitHub, Inc., “GitHub.” URL: https://github.com/

[324] LinkedIn Corporation, “LinkedIn.” URL: https://www.linkedin.com

[325] The Matrix.org Foundation C.I.C., “Matrix.” URL: https://matrix.org/

[326] Reddit, “Reddit.” URL: https://www.reddit.com/

[327] Stack Exchange, Inc., “Stack Overflow.” URL: https://stackoverflow.com/

[328] Telegram, “Telegram.” URL: https://telegram.org/

[329] Twitter, Inc., “Twitter.” URL: https://twitter.com/

[330] Google, LLC, “YouTube.” URL: https://www.youtube.com/

[331] OpenAPI Tools, “OpenAPI Generator.” URL: https://github.com/OpenAPITools/openapi-generator

[332] Object Modeling Group, “Unified modeling language specification.” URL: https://www.omg.org/spec/UML

[333] A. Danial, “cloc: v1.92,” 2021. URL: https://doi.org/10.5281/zenodo.5760077

[334] Oceane Duboust, “Elon Musk’s X sheds the last of its Twitter branding by changing web address to x.com.” URL: https://www.euronews.com/next/2024/05/18/elon-musks-x-sheds-the-last-of-its-twitter-branding-by-changing-web-address-to-xcom