Using Crowd-Based Software Repositories to Better Understand Developer-User Interactions

by

Wenhan Zhu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2023

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Massimiliano Di Penta Professor, Department of Engineering University of Sannio
Supervisor(s):	Michael W. Godfrey Professor, David R. Cheriton School of Computer Science University of Waterloo
Internal Member:	Shane McIntosh Associate Professor, David R. Cheriton School of Computer Science University of Waterloo
	Chengnian Sun Assistant Professor, David R. Cheriton School of Computer Science University of Waterloo
Internal-External Member:	Derek Rayside Associate Professor, Electrical and Computer Engineering University of Waterloo

Author's declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Throughout this thesis, I served as the primary author for all the research presented. My responsibilities include (1) developing the concept of the study (2) collecting data and performing analysis (3) writing the draft manuscript. My co-authors helped me with refining the research ideas, providing feedback at each step of the research process and improving the manuscript.

In the research presented in Chapter 4, Haoxiang Zhang mentored me on how to perform an empirical study. He helped with the data collection and analysis design of the project. Ahmed E. Hassan and Michael W. Godfrey provided feedback and supervision in all parts of the research.

In the research presented in Chapter 5, Michael W. Godfrey provided feedback and supervision in all parts of the research. He also helped with the qualitative part of the study.

In the research presented in Chapter 6, Sebastian Proksch, Daniel M. German, Michael W. Godfrey, Li Li, Shane McIntosh participated in the qualitative part of the study and provided feedback in all parts of the research. Sebastian Proksch helped with the data collection and labeling process. Daniel M. German helped with improving the structure and coherence of the manuscript. Michael W. Godfrey, Sebastian Proksch, and Daniel M. German provided supervision in all parts of the research.

Abstract

Software development is a complex process. To serve the final software product to the end user, developers need to rely on a variety of software artifacts throughout the development process. The term *software repository* used to denote only containers of source code such as version control systems; more recent usage has generalized the concept to include a plethora of software development artifact kinds and their related meta-data.

Broadly speaking, software repositories include version control systems, technical documentation, issue trackers, question and answer sites, distribution information, etc. The software repositories can be based on a specific project (e.g., bug tracker for Firefox), or be crowd-sourced (e.g., questions and answers on technical Q&A websites). Crowd-based software artifacts are created as by-products of developer-user interactions which are sometimes referred to as communication channels. In this thesis, we investigate three distinct crowd-based software repositories that follow different models of developer-user interactions. We believe through a better understanding of the crowd-based software repositories, we can identify challenges in software development and provide insights to improve the software development process.

In our first study, we investigate Stack Overflow. It is the largest collection of programming related questions and answers. On Stack Overflow, developers interact with other developers to create crowd-sourced knowledge in the form of questions and answers. The results of the interactions (i.e., the question threads) become valuable information to the entire developer community. Prior research on Stack Overflow tacitly assume that questions receives answers directly on the platform and no need of interaction is required during the process. Meanwhile, the platform allows attaching comments to questions which forms discussions of the question. Our study found that question discussions occur for 59.2% of questions on Stack Overflow. For discussed and solved questions on Stack Overflow, 80.6% of the questions have the discussion begin before the accepted answer is submitted. The results of our study show the importance and nuances of interactions in technical Q&A.

We then study *dotfiles*, a set of publicly shared user-specific configuration files for software tools. There is a culture of sharing *dotfiles* within the developer community, where the idea is to learn from other developers' *dotfiles* and share your variants. The interaction of *dotfiles* sharing can be viewed as developers sources information from other developers, adapt the information to their own needs, and share their adaptations back to the community. Our study on *dotfiles* suggests that is a common practice among developers to share *dotfiles* where 25.8% of the most stared users on *GitHub* have a *dotfiles* repository. We provide a taxonomy of the commonly tracked *dotfiles* and a qualitative study on the

commits in *dotfiles* repositories. We also leveraged the state-of-the-art time-series clustering technique (K-shape) to identify code churn pattern for *dotfile* edits. This study is the first step towards understanding the practices of maintaining and sharing *dotfiles*.

Finally, we study app stores, the platforms that distribute software products and contain many non-technical attributes (e.g., ratings and reviews) of software products. Three major stakeholders interacts with each other in app stores: the app store owner who governs the operation of the app store; developers who publish applications on the app store; and users who browse and download applications in the app store. App stores often provide means of interaction between all three actors (e.g., app reviews, store policy) and sometimes interactions with in the same actor (e.g., developer forum). We surveyed existing app stores to extract key features from app store operation. We then labeled a representative set of app store collected by web queries. K-means is applied to the labeled app stores to detect natural groupings of app stores. We observed a diverse set of app stores through the process. Instead of a single model that describes all app stores, fundamentally, our observations show that app stores operates differently. This study provide insights in understanding how app stores can affect software development.

In summary, we investigated software repositories containing software artifacts created from different developer-user interactions. These software repositories are essential for software development in providing referencing information (i.e., Stack Overflow), improving development productivity (i.e., *dotfiles*), and help distributing the software products to end users (i.e., app stores).

Acknowledgements

I am deeply grateful to all the individuals who have supported me throughout my PhD studies. Their guidance and encouragement have been instrumental in my journey as a researcher in the field of software engineering.

First and foremost, I would like to express my sincere appreciation to my supervisor, Michael W. Godfrey. He took me in as his student when I was unsure of my fate as a PhD student, and guided me towards becoming a proficient researcher. Mike has been incredibly supportive throughout the process, providing me with the freedom to explore my interests and offering insightful feedback on my work. I am also thankful to Meiyappan Nagappan, who stepped in and acted as my supervisor when Mike was dealing with personal issues. I appreciated the thought-provoking discussions we had during this period.

Furthermore, I would like to thank Krzystof Czarnecki for accepting me as a PhD student at the University of Waterloo. Despite our mismatched research interests, I am grateful for the opportunity to work in his lab and learn about the cutting-edge developments in the field of autonomous vehicles.

I would like to express my gratitude to my examination committee members — Shane McIntosh, Chengnian Sun, Derek Rayside, and Massimiliano Di Penta — for their time and effort in providing feedback on this thesis.

In addition, I am fortunate to have worked with many collaborators, including Haoxiang Zhang, Ahmed E. Hassan, Sebastian Proksch, Daniel M. German, Li Li, and Shane McIntosh. They provided valuable feedback on my research and helped me improve in the process. I am especially grateful to Haoxiang Zhang for his hands-on mentoring to finish my first research paper and for offering me an opportunity for an internship that exposed me to industrial research.

I would also like to thank the members of the Waterloo Intelligent Systems Engineering Lab (WISE) and the Software Analytics Group (SWAG), where I had the privilege of staying during my PhD studies. In particular, I would like to thank Changjian Li, Jaeyoung Lee, Edward Chao, Weitao (Tommy) Chen, Jian Deng from WISE, and Yiwen Dong, Yongqiang Tian, Daniel J. Watson, Kilby Baaron, Davood Anbarnam, Kirsten (Ten) Bradley, Reza Nadri, Achyudh Ram, Lakshmanan Arumugam, Arman Naeimian, Aaron Sarson, Bushra Aloraini, Farshad Kazemi, Zhenyang Xu, Yaxin Cheng, Xueyao (Eve) Yu, Zhili Zeng, and Gengyi Sun from SWAG. I apologize if I have missed anyone's name due to the length of my PhD study and the number of people I have met.

I would like to express my sincere gratitude to my friends and roommates, Yusen Su and Xizi (Lucy) Wang, who generously added a pair of chopsticks during meals to feed me.

Lastly, I would like to thank my parents — Weijun Zhu and Yi Lian — for their unconditional love and support. Especially for the year and a half of remote work due to the COVID-19 pandemic, where I returned home and lived with them, I am grateful for the time we spent together. Their unwavering encouragement and belief in me has been a source of motivation and strength throughout my PhD journey.

Dedication

This thesis is dedicated to those who look up into the sky.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The process of developing software products is complex and requires a variety of resources. These resources, as well as the by-products of the software development process, are commonly referred to as software artifacts. A collection of software artifacts will form a software repository. While the term "software repository" was originally used to describe containers of source code exclusively, more recent usage has generalized the concept to include information contained in different kinds of software artifacts and their related meta-data.

Software repositories include datasets and information related to artifacts created during the software development process. These artifacts may include API documentation, bug reports, release notes, and user reviews. Increasingly, researchers recognize that these sources of information are known to be crucial to the success of software development. For example, the quality of comments in code can directly impact the effort required to maintain it [44]. With the creation and adoption of new tools; the establishment of common practices, and the standardization of approaches, we have seen a plethora of new software repositories derived and formed through developer activities.

Software repositories can also arise from interactions between developers and users. Despite the source code being the end product of software development, the development process relies on teams of developers who are often influenced by social activities in software engineering. The information generated from social interactions provides rich data for both developers and researchers, making it a valuable resource for understanding and improving the development process. For example, issue trackers have been shown to be essential in fixing bugs and tracking feature implementation [17, 19]. Through studying the existing data in issue trackers, researchers have been able to identify key challenges and provide insights on how to improve the process [16, 87, 15].

1.1 Five Types of Software Repositories

The software development process involves various types of software artifacts that developers encounter. We categorize these artifacts into five types, which are presented in Fig. 1.1.

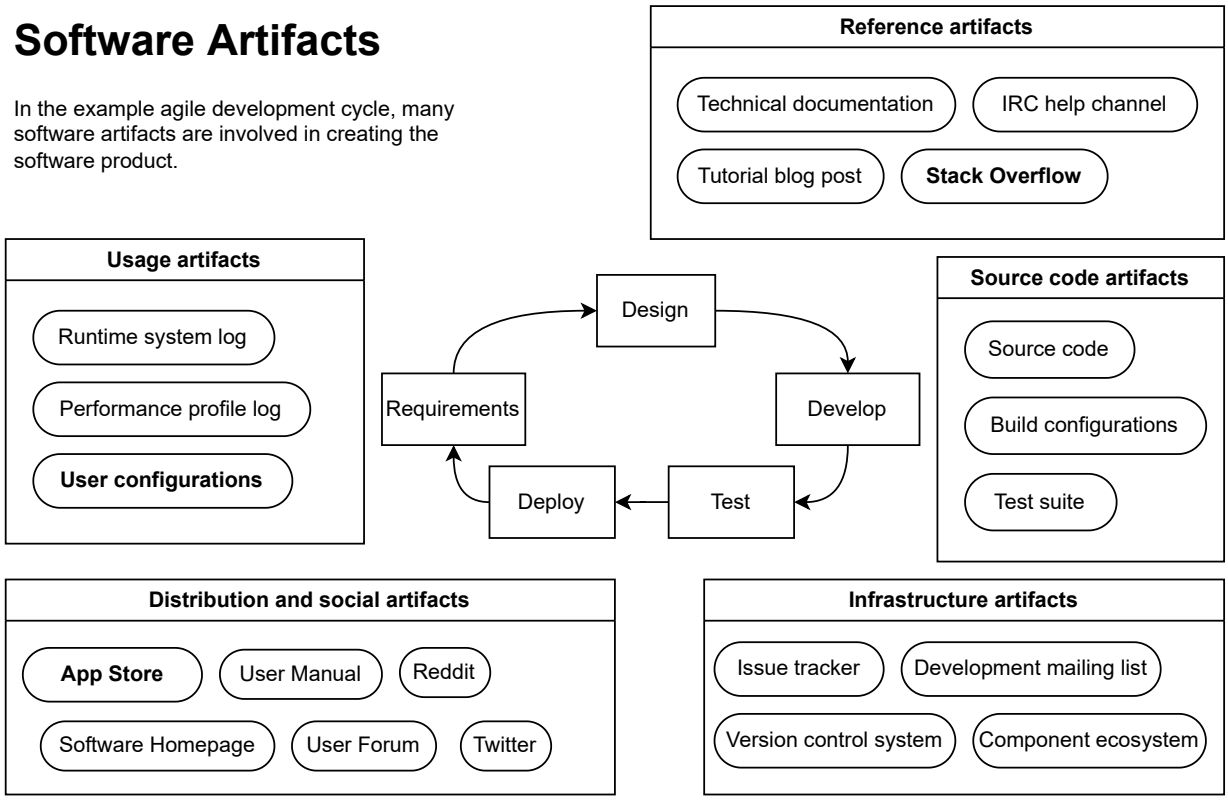


Figure 1.1: Software repositories and the agile development cycle

Specifically, the five types of software artifacts is described as follows.

- **Source code artifacts** — These artifacts directly contribute to the production of the final software product, including source code, building scripts, test suites, and deployment and configuration tools.
- **Infrastructure artifacts** — These artifacts are relate to or by-products of the ongoing software development processes, such as issue tracking systems, version control

systems, and project-specific development mailing lists. They provide a record of the design decisions and evolution history of the source code.

- **Reference artifacts** — These artifacts contain general information that developers need during software development, such as technical documentation, tutorial blog post, user forums, and Stack Overflow. Developers consult these software repositories to resolve their issues during software development.

- **Usage artifacts** — These artifacts contain system usage information, such as user configuration files, diagnostic, system logs, performance data, and other instrumentation information that can be used to infer user behavioural patterns. These artifacts reflect how the software systems run in real world scenarios.

- **Deployment and social artifacts** — These artifacts contribute to the final distribution of the software system to its designated users. Examples include software homepage and app store. Software systems are often designed to solve real-world problems, and the deployment artifacts connects the software systems to their potential users.

The same artifact may belong to different types, depending on the perspective and platform. For example, when a user requests help debugging a specific configuration for a software tool on a social platform like Reddit, the content could fall under three different types of software artifacts. Specifically, the configuration is a usage artifact, the ensuing thread becomes a reference artifact for future viewers, and the interaction between the asker and respondents forms part of the distribution and social artifact where end users discuss the software.

The software artifacts are created to satisfy the requirements of both developers and users in software development. This process is also dynamic, with rudimentary forms emerging initially and being improved over time, based on practical evidence, to better serve developers' needs. For example, the Linux kernel development team identified the need for an enhanced version control system and created *git*, which eventually became the most widely used version control tool. Prior to *git*, different tools and ideologies were used in maintaining version control information, reflecting the evolving nature of software artifacts over time.

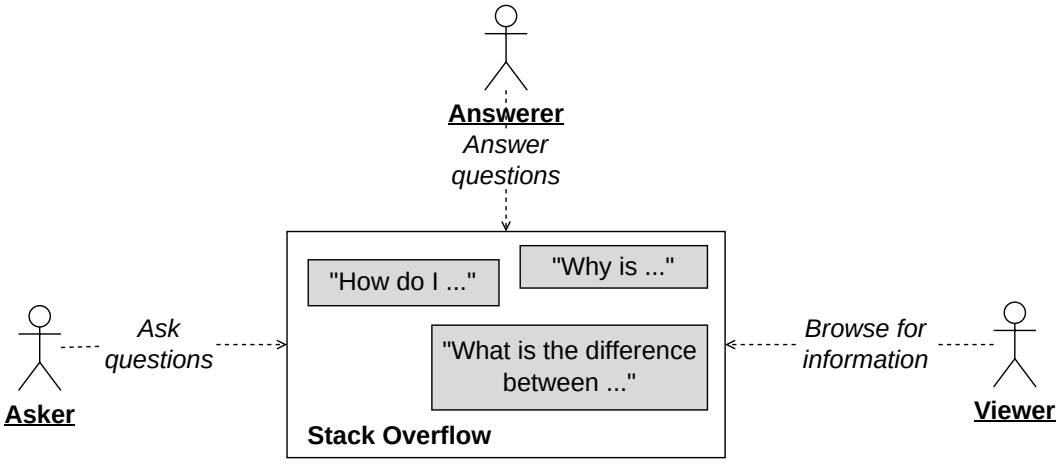
1.2 Software Repositories from Developer-User Interactions

Among the broad spectrum of software repositories, software artifacts from developer-user interactions are often distinct in how they are created and leveraged. This is especially true when the software artifacts are created as the by-products of the interactions instead of being created intentionally. For example, let us take as an example *question threads*, which are software artifacts on Stack Overflow. The interaction between various users aims to provide an answer to the question that has been posed by the asker. The interaction is finished when the asker judges that an answer has answered the question satisfactorily. Both the asker and answerer conclude the interaction at this moment. However, the resulting knowledge created from the interactions remains beneficial to all viewers who may encounter a similar question in the future. Programming-related Q&A has a long history even before Stack Overflow; however, the structure of preserving the Q&A session in question threads contributed to the success of Stack Overflow. In crowd-based software repositories, the by-products are often beneficial to a broader audience in addition to the people involved in the original interaction that generated the software artifact.

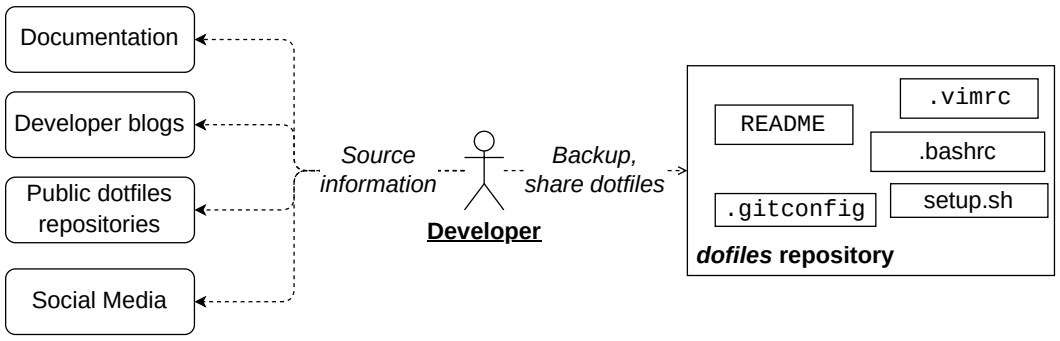
In this thesis, we study three software repositories crucial to the software development process that are created from crowd developer-user interactions. These three software repositories represent different patterns of interaction models. Through exploring the distinct model of interactions, we can gain a better understanding of the software development process, and how to better leverage the information contained in the resulting software artifacts. Specifically, we focus on Stack Overflow: a collection of programming question and answers; *dotfiles*: a collection of user-specific configuration files; and app stores: the target distribution platform for software products which contains many non-technical attributes that cannot be directly derived from the software products.

The three software repositories we examine represents three different developer-user interaction models, as illustrated in Fig. 1.2.

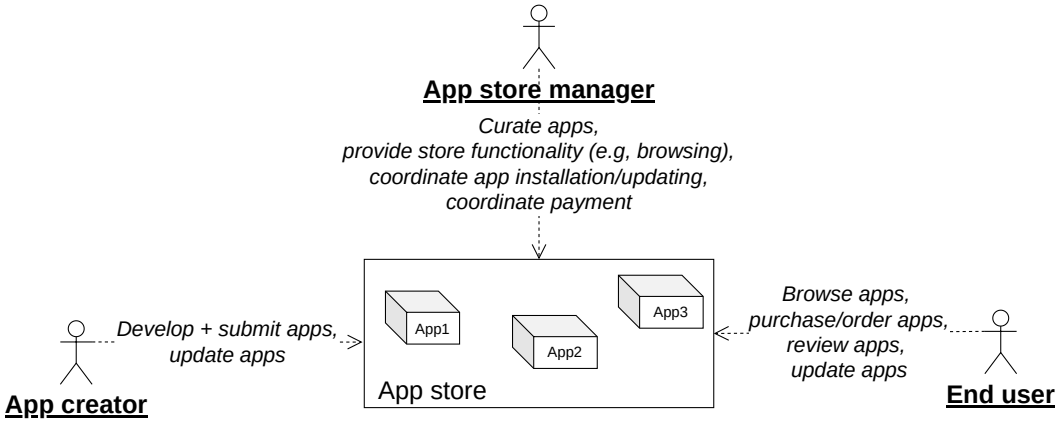
Stack Overflow is a technical Q&A website that has become the largest knowledge base for programming related questions and answers. There are two main type of interactions involved on Stack Overflow: the process of creating the unit of knowledge (i.e., a question thread), and users viewing the website for its existing knowledge. This relationship is shown in Fig. 1.2a. As direct contributions to online information are rare on many platforms [161, 103], understanding the creation and maintenance of knowledge on Stack Overflow is crucial. In this thesis, we focus on the question discussions that occur on the platform to better understand the Q&A process.



(a) Stack Overflow developer-user interaction model



(b) *dotfiles* developer-user interaction model



(c) App store developer-user interaction model

Figure 1.2: Three types of developer-user interaction

dotfiles are software artifacts that represents real-world configurations for user-specific software tools. The process of how developers create and share *dotfiles* is a great example of "one for all, all for one". When developers first write their configurations, they source from various resources. Then, after absorbing the information and creating their own adaptions, developers share back their own additions, benefiting the community as a whole. This process is shown in Fig. 1.2b. In this thesis, we study a collection of *dotfiles* repositories from *GitHub* to understand how developers maintain their *dotfiles*.

App stores serve as distribution platforms where developers can submit their applications. The successful operation of these stores involves three major stakeholders: *store owners* who oversee the app store, *developers* who contribute applications, and *users* who browse and download them. Fig. 1.2c illustrates the interactions between these parties within the app store platform, which offers ample opportunities for interactions. For example, app stores often feature product-specific homepages for developers to showcase their applications to potential users, as well as review sections where users can share their experiences of the app with other users and the developers. In this thesis, we explore the broader landscape of app stores beyond the mobile-focused scene, aiming to understand their functionality, diversity, and how their existence affects software development practices.

1.3 Thesis Statement

Thesis Statement: *Studying crowd-based software repositories can enhance our understanding of developer-user interactions during the creation of software artifacts. With this deeper understanding, we can identify challenges in software development and provide valuable insights to improve the software development process.*

The thesis statement is supported by three studies on different software repositories, each of which contributes to our understanding of developer-user interactions in software development. The three studies are as follows:

- **Understanding question discussions on Stack Overflow**

  Developers consults a variety of resources for information during software development, including technical Q&A websites like Stack Overflow. Most previous research assumed that questions received answers directly on the platform and that no interaction was required during the process. However, recent research have suggested that answer comments can provide valuable information for users seeking answers on

the platform. In this research, we studied the commenting activities for questions on Stack Overflow and gained insights into the Q&A process on the platform. We find that participation in question discussions is common among askers and answerers on the platform, and that the interactive nature of Q&A sessions is an important aspect of technical Q&A.

- **Understanding user-specific configuration files**

  Developers often use tools to simplify their jobs and boost productivity. These tools provide high customizability, allowing developers to create configurations for the software that match their preferences. These configurations are often stored in plain-text files known as *dotfiles*, which are shared publicly in the developer community. In this study, we examined publicly shared *dotfiles* on *GitHub* to understand the developer interactions in maintaining *dotfiles*. We find that the practice is popular among prolific developers on *GitHub*, and we identified a taxonomy of common *dotfile* types to understand the types of configurations developers manage in their *dotfiles*. We also identified the intents of commits in *dotfiles* repositories, which include both tweaking user-specific configuration and managing the *dotfiles* repository's documentation and deployment.

- **Understanding the role of app stores in the perspective of software engineering**

  App stores have become the primary destination for software distribution. Software products are developed and submitted to app stores, where users can browse and install them with ease. While the concept of app stores is popularized by the two major mobile stores (i.e., Apple's App Store and Google Play), many other app stores exist, each offering different features. In this study, we performed an empirical study on app stores to understand the concept of app stores and the commonalities and differences between them. We provide a working definition of app stores and identified a set of features that can be used to describe app stores. We used these features to label a set of app stores collected from web queries and demonstrated the existence of natural groupings based on the features offered by the app stores.

1.4 Thesis Contributions

This dissertation has seven main contributions:

- We provide an extended view of the process of creating answers on Stack Overflow with the inclusion of question discussions (Chapter 4). In most of the answered questions that undertake discussion, the discussion began before any answer was provided. Participation in question discussions is common among askers and answerers on the platform.

- We highlight the importance of interactions during technical Q&A (Chapter 4). While in ideal cases, questions should have direct answers, in practice, users often need to go through an interactive session to solve the question.

- We explored the concept of sharing user-specific configuration files among developers (Chapter 5). We show that the practice is popular among prolific developers on *GitHub*, and checked the occupations of shared *dotfiles* repository owners.

- We provide a taxonomy on the type of common *dotfiles* (Chapter 5). From common *dotfiles* in *dotfiles* repositories, we group the *dotfiles* by their associated software and functionality. The taxonomy allows us to understand the type of configurations developer manage in their *dotfiles* repositories.

- We identified intents of commits in *dotfiles* repositories (Chapter 5). While most of the commits directly focus on tweaking user-specific configuration, we also observed a significant amount of commits focusing on managing the *dotfiles* repository in its documentation and deployment.

- We provide a working definition of app stores, expanding upon existing understandings on the concept of app stores (Chapter 6). While the concept of app stores is popularized by the two mobile stores (i.e., Apple's App Store and Google Play), there exists a diverse set of app stores following different practices.

- We identified a set of features that can be used to describe app stores (Chapter 6). We use the features to label a set of app stores collected from web queries. We then demonstrate the existence of a natural grouping based on the features offered by app stores.

1.5 Organization

The rest of the thesis is organized as the following: Chapter 2 covers the background information of the software artifacts studied in this thesis. Chapter 3 discusses the related

work on topic of this thesis and provides a overview of the existing understanding of the studied software repositories. Chapter 4 describes our study on question discussions on Stack Overflow. Chapter 5 shows our exploration on how developers manage their *dotfiles*. Chapter 6 presents our study to understanding the concept of app stores in software engineering. Chapter 7 summarized the results of this thesis and discusses some possible future research directions.

Chapter 2

Background

In this chapter, we summarize and discuss previous research that relates to our work in the three software repositories.

2.1 Knowledge Creation on Stack Overflow

Stack Overflow is a website dedicated to programming related questions and answers. Users can ask questions and provide answers on the platform. As a Q&A platform, it has become a primary communication channel for developers seeking programming-related assistance. The success of Stack Overflow can largely be attributed to its gamification system, whereby users earn reputation points when their questions or answers receive endorsement from other users via a voting system. The website has been widely embraced by the software engineering community, and has grown to become the largest public knowledge base for programming-related questions, featuring 21.9 million questions and 32.7 million answers as of December 2021.

2.1.1 Question and Answer (Q&A) Process

At the core of Stack Overflow is the individual questions and answers that contribute to the large collection of crowdsourced knowledge. The starting point for accessing this knowledge is a question post, in which a user submits a question pertaining to programming or a similar technical topic. At that point, other users can start to engage either by proposing an *answer*, or by taking part in a *discussion* in the form of a *comment* or a *chat room*.

Discussions can be attached to either the original question (i.e., a *question discussion*) or one of the proposed answers (i.e., an *answer discussion*). If a proposed answer successfully resolves the question, the user who asked the original question (i.e., the *asker*) may at their discretion choose to designate that answer as the *accepted answer*. In other words, the accepted answer is what the asker considered as the answer that solved the question. Once an accepted answer has been selected, users may continue to contribute to the question thread by adding new answers or editing existing content; in practice, however, user activity related to that question and its answers tends to diminish sharply at that point [14]. In practice, as shown both by community wisdom[1], and research effort [51], the accepted answer may not be the best answer for the specific question. We note that the Stack Overflow uses the term *post* internally to refer to either a question or answer, but not a discussion.

2.1.2 Discussions on Stack Overflow

Stack Overflow offers two different forms of communication channels for users to discuss on questions and answers, that is, commenting as an asynchronous communication channel and chatting as a synchronous communication channel. When users are commenting, they may not expect an immediate reply. Meanwhile, when users are chatting, a live session is formed where information flows freely within the group in real-time [139]. On Stack Overflow, users begin discussions in comments. When extended discussions occur in comments, users are proposed with continuing the discussions in dedicated chat rooms. While commenting is the dominating communication channel on the Stack Overflow for question discussions, whenever possible, we take special notice of the existence of chat rooms since they represent a different form of communication channel compared to comments.

As previously mentioned, users can attach comments to a post (i.e., a question or answer). Stack Overflow considers comments as *"temporary 'Post-It' notes left on a question or answer.*[2]*"* Stack Overflow users are encouraged to post comments *"to request clarification from the author; leave constructive criticism to guide the author in improving the post, and add relevant but minor or transient information to a post."* When multiple comments are present in the same post, they form a *comment thread*.

Stack Overflow offers *real-time, persistent collaborative chat for the community*[3] with chat rooms. Stack Overflow promotes users to continue the discussions in a chat room

[1] https://meta.stackoverflow.com/questions/283456
[2] https://stackoverflow.com/help/privileges/comment
[3] https://chat.stackoverflow.com/faq

when there are more than three back-and-forth comments between two users (i.e., at least 6 in total). Users are prompted with a message before a chat room can be created: *"Please avoid extended discussions in comments. Would you like to automatically move this discussion to chat?"* When the user agrees to create the chat room, an automated comment is posted and contains a link to the newly created chat room. In the newly created chat room, automated messages are posted indicating the associated question and the comments leading to the chat room. Users can also create chat rooms directly that are not associated with questions or answers. Figure 2.1 shows an example question discussion session that started in comments and concluded in a chat room. The question discussion eventually led to the creation of an answer that was later marked by the asker as the accepted answer.

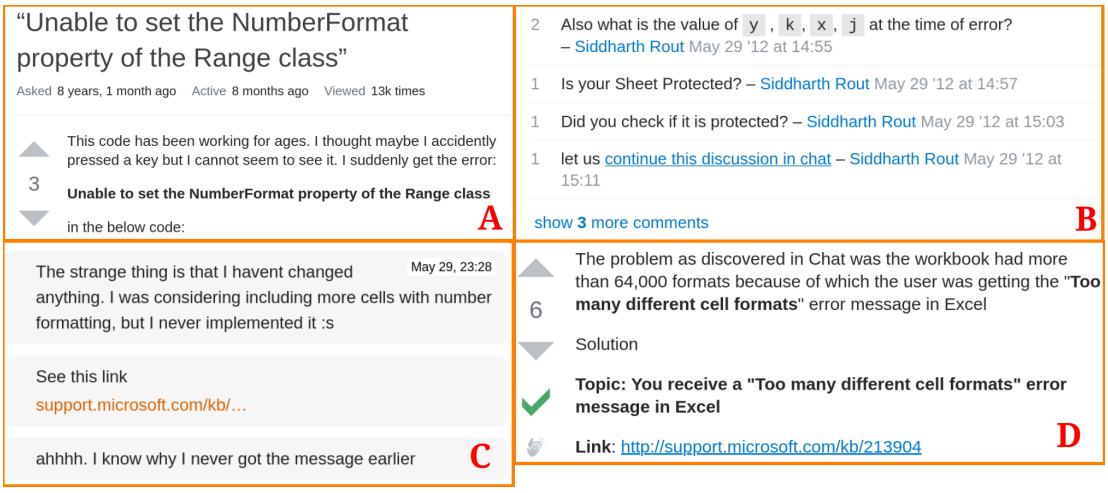


Figure 2.1: Stack Overflow question thread with discussion in comments and chat room messages

2.2 User-Specific Configuration Files — *dotfiles*

We use to the term "dotfiles" to refer to a collection of user-specific configuration files [160]. The term *dotfiles* in its original meaning refers to the hidden files in UNIX-like (**NIX*) operating systems. The concept originates from a bug in an early implementation of the command *ls* [108], which produces a listing of the files in a given directory. The bug occurred when the code — correctly — ignored the two special files that represent the current and parent directory (i.e., "." and ".."), but also — incorrectly — ignored all other files that started with a dot. Subsequently, a convention arose within the **NIX*

community to prepend a period onto the beginning of file names that store application settings in users' home directories; these files would be "hidden" by default from the user's view unless they explicitly asked to see the *dotfiles*, via the `-a` option to `ls`. "Hiding" the configuration files in this way reduces visible clutter in the user's home directory, and reduces the risk of new users accidentally changing or removing settings files that they may not understand well. With the wide spread adoption of storing configurations in *dotfiles*, the meaning of the term have since evolved to refer to the collection of user-specific configuration files.

2.2.1 Culture of Sharing *dotfiles*

As suggested by a community website dedicated to *dotfiles* hosted on *GitHub* [42], users backup their *dotfiles* online, learn from the existing *dotfiles* from community, and share back what they have learned to the community. While it is unlikely that there is enough evidence to trace the first occurrence of *dotfiles* sharing, the activity can be observed at least as far back as the early 1980s, when *USENET* and e-mailing lists served as proto-social media for developers [131]. Users often share their personal configuration and wisdom for a program where others can benefit from; this culture of sharing has fostered a vibrant user community. One blog post stood out that brought the idea to a larger community: Holman, one of the first engineers at *GitHub*, wrote a impactful blog entry titled "Dotfiles are meant to be forked" [64]. His *dotfiles* repository [64] is also one of the most starred *dotfiles* repository on *GitHub*. With *GitHub* gaining more popularity and becoming the largest source code hosting platform, many developers have hosted their *dotfiles* repository on *GitHub*.

2.3 App Stores

Wikipedia recognizes Electronic AppWrapper [162] as the first true platform-specific electronic marketplace for software applications, but the term became popular when Apple introduced its App Store along with the iPhone 3G in 2008 [10]. Today Apple's App Store and Google Play are the two biggest app stores and both of them serve the mobile market. At its essence, app store is a type of digital distribution platform for software products. Despite app stores often tacitly refers to the two mobile stores, in practice, a variety of app stores exists and serve different users.

2.3.1 App Store Operation

The operation of the app store involves three major stakeholders. The app store owner who manages the app store; the developers who submit applications to the store; and the users who browse and install applications from the app store.

To ensure the successful operation of the app store, many features exist within the app store to support each major stakeholder. For example, a developer portal where developers can view marketing feedback from their applications and user reviews where users can share their experience with the application with other users and the developer.

Chapter 3

Related Work

In this chapter, we discuss about the related work to the topics included in this thesis. Specifically, we discuss how the related work increases our understanding of the studied software repositories, and how prior research were able to leverage the software repositories to improve the software development process.

3.1 Exploring and Understanding Stack Overflow

Since the introduction of Stack Overflow to the developer community, it has become the *de facto* source for programming related Q&A. Stack Overflow has a friendly license[1] for information reuse. Moreover, Stack Exchange (Stack Overflow's parent company) releases a site data dump periodically. The conveniences allow researchers to study the information contained on the website easily. A large amount of effort has been spent by researchers on Stack Overflow to understand its success in sharing knowledge and also to leverage its large corpus of data to aid the software development process.

3.1.1 Discussion Activities on Stack Overflow

While Stack Overflow is mainly a Q&A platform, in addition to question and answering, it also has many other mechanisms to help with the Q&A process (e.g., the gamification system through reputation points and commenting). In this thesis, we consider comments

[1] https://stackoverflow.com/help/licensing

associated with questions as question discussions. However, in many other works, a discussion on Stack Overflow can have different meanings. For example, some studies [81, 128] have considered the question as a discussion (e.g., the question, all its associated answers, and any comment associated with the question or its answers). In Chapter 4, we use discussions to describe commenting activities associated with a specific post (i.e., a question or an answer).

Most previous works on Stack Overflow discussions have a primary focus on answer discussions. Their aim is to better understand the community efforts in improving the crowdsourced knowledge on Stack Overflow. Zhang et al. [179] conducted an empirical study to understand answer obsolescence on Stack Overflow. In their study, comments are used as an indicator of obsolescence for their associated answer. A follow up study by Zhang et al. [178] examined answer comments and highlighted that the information contained in the comments should not be overlooked when reading their associated answers. After acknowledging the importance of answers, Zhang et al. [180] focused on the current commenting mechanism on Stack Overflow and observed that the current presentation of comment information is not optimal for readers. The comment hiding mechanism on Stack Overflow only displays the top five comments with the most upvotes. However, due to most comments never receiving any upvotes, later comments, which are likely to be more informative, are hidden from readers by default.

Comments are also viewed as triggers for post updates. Baltes et al. [14] observed that post edits often occur shortly after comment posts and suggests that comments and post edits are closely related. Based on this observation, a study by Soni et al. [135] further analyzed how comments affect answer updates on Stack Overflow. Their observation echoes the finding by Zhang et al. [179] that unfortunately users do not update their answers even with comments directly suggesting so.

Compared to existing studies on Stack Overflow which mostly focus on answers from the perspective of knowledge maintenance, in this thesis, we focus on the question discussions that mainly begin and occur during the Q&A process. In other words, previous works have focused on preserving the knowledge while our work tends to focus more on the creation of the knowledge.

3.1.2 Leveraging Discussions in Software Engineering

During software development, communication between members of the team is important for the long-term success of the project [139]. Online discussions are a core part of the process, especially in open source projects where developers may be scattered around the

world and rely on a variety of channels to communicate with each other [139]. Di Sorbo et al. [37] proposed an NLP based approach to mine developer communication channels and automatically classify the type of information (e.g., feature request, bug). Since the advent of ubiquitous e-mail in the 1980s, developers have used mailing lists for discussions about the projects they are working on and interested in. Studies show that the use of mailing lists facilitates the gathering of people with similar interests, and many open source projects still run mailing lists today [129] (e.g., the Gnome mailing list[2]). The mailing list archive is an informative resource for researchers to understand the development of the project. Rigby et al. [118] studied the Apache developer mailing list to learn about the personality traits of developers and how the traits shift during the development of the project. Sowe et al. [136] studied three Debian mailing lists and constructed social networks of the mailing list to investigate how knowledge is shared between expert to novice participants.

In addition to the asynchronous email exchanges, developers also use real-time communication channels such as IRC for discussions. IRC channels are often used by open source projects as a complement to their mailing list operations (e.g., the `#emacs` channel on Freenode exists in addition to the project's mailing list). Shihab et al. investigated *GNOME GTK+* [129, 130] and *Evolution* [130] IRC channels to better understand how developers discuss in IRC. Although e-mail and IRC are still in use today, newer and more efficient platforms have also emerged to better support the need for communication. For example, developers report bugs and feature requests on issue trackers (e.g., Jira[3]), and ask questions on Stack Overflow [150]. Vasilescu et al. [150] observed that in the *R* community, developers are moving away from the *r-help* mailing list to sites like Stack Overflow in the Stack Exchange network since questions are answered faster there. Prior studies examined different communication channels aiming to better understand and improve the communication among developers. Alkadhi et al. [4] applied content analysis and machine learning techniques to extract the rationale from chat messages to better understand the developers' intent and the decision making process during software development. Lin et al. [79] studied the usage of Slack by developers and noticed that bots are in discussions to help software developers.

Storey et al. [139] surveyed how developers leveraged communication channels and observed that real-time messaging tools and Q&A platforms such as Stack Overflow are essential for developing software. Dittrich et al. [38] studied developers' communication across different platforms and observed that real-time messaging plays a role in the communication of developers. Their study shows that real-time messaging tools can support the usage of other communication channels (e.g., Skype calls) and provide a means for de-

[2] https://mail.gnome.org/mailman/listinfo

[3] https://www.atlassian.com/software/jira

velopers to form social and trust relationships with their colleagues. Chatterjee et al. [24] analyzed characteristics of Q&A sessions in Slack and observed that they cover the same topics as Stack Overflow. Wei et al. [184] applied neural networks techniques on real-time messages to automatically capture Q&A sessions. Ford et al. [49] experimented with using real-time chat rooms for the mentoring of asking questions on Stack Overflow for novice users. Chowdhury et al. [31] leveraged information from Stack Overflow to create a content filter to effectively filter irrelevant discussions in IRC channels.

3.1.3 Understanding and Improving Stack Overflow

Prior research investigated how developers leverage Stack Overflow and studied different mechanisms aiming to improve the design of Stack Overflow [166, 25, 186, 156, 49]. Treude et al. [145] categorized the types of questions on Stack Overflow, and observed that Stack Overflow can be useful for code review and learning the concepts of programming. Wang et al. [156] studied the edits of answers and observed that users leverage the gamification system on Stack Overflow to gain more reputation points. Prior studies also aimed to understand the quality of the crowdsourced knowledge on Stack Overflow. Ponzanelli et al. [109] proposed a method to classify questions according to their quality-based features extracted from question body. Srba et al. [138] observed that an increasing amount of content with relatively lower quality is affecting the Stack Overflow community. Lower quality content on Stack Overflow may also affect how questions are answered. Asaduzszaman et al. [12] showed that the quality of questions plays an important role in whether a question receives an answer by studying unanswered questions on Stack Overflow. An automated system to identify the quality of posts and filter low-quality content was proposed by Ponzanelli et al. [110]. To improve the quality of the crowdsourced knowledge on Stack Overflow, prior studies aimed to identify artifacts with different properties [157, 111, 30, 144, 150, 172, 186]. For example, Nasehi et al. [96] examined code examples on Stack Overflow and identified characteristics of effective code examples. Their study shows that explanations for code examples have the same importance as code examples. Yang et al. [170] analyzed code snippets of popular languages (C#, Java, JavaScript, and Python) on Stack Overflow and examined their usability by compiling or running them.

Prior studies also examined various supporting processes on Stack Overflow to better understand its operation and improve its efficiency of the crowdsourced knowledge sharing process. Chen et al. [25] used a convolutional neural network (CNN) based approach to predict the need for post revisions to improve the overall quality of Stack Overflow posts. Several studies proposed approaches to automatically predict tags on Stack Overflow [166,

122, 18]. Wang et al. [158, 159] proposed an automatic recommender for tags based on historical tag assignments to improve the accuracy of the labeling of tags for questions.

3.2 Configuration Files

Most of existing research in this broad area focuses on studying software configurations either during the build process (e.g., enabling different features for different configurations of software), or on the configuration of essential production software (e.g., database systems, web servers). In this thesis, our focus is on the user-specific configuration files that users keep to configure the software they use. Since they are both configuration files, and more focused on the scene of developer life, in this section, we provide a overview of related work in both software configuration, and improving developer workflow.

3.2.1 Software Configurations

While many studies have looked at configurations, the main focus has been on software build configurations and software run-time configurations [94, 187, 95, 167, 168, 185]. These studies often focus on production software and the aim is to improve the build process or optimize production software performance. This includes studying the build configuration files. Nadi et al. [94] proposed a static analysis approach to extract configuration constraints from software build files (e.g., `Makefile`s). Their methods are able to provide insight from creating a variability model from the constraints and reason about the build configuration. Zhou et al. [187] proposed a symbolic based extraction method to parse build files such as `Makefile`s and detect potential problems in the configuration space for software. Other research focuses on run-time configurations. Many software systems that are run in production are highly customizable with many parameters to fit different needs. Unlike build-time configurations which cannot be changed after the software is built, run-time configurations are more flexible. Databases are a class of software that fits the description well, and it is no surprise that they are also one of the most studied systems for run-time configurations [43]. One goal of tweaking run-time configurations is to have the software system running at optimal settings. Nair et al. [95] proposed a sequential model-based method that explores the configuration space and try to determine the next best configuration. Mühlbauer et al. [90] showed that Gaussian Process models can accurately estimate the performance-evolution history of real-world software systems. Kaltenecker et al. [73] showed that from empirical evaluation, distance-based sampling

on configuration space can yield more accurate performance models for medium to large sample sets.

Studies have also shown that the design of configuration "knobs" are suboptimal. Xu et al. [167] investigated on the complexity of configuration settings on multiple software systems. Their study shows that only a small percentage of configurations are altered by users while a significant percentage of the configurations are never changed. The complexity of configurations can also cause problems on the user side, where users do not fully understand the functionalities of the configurations. Based on their results, a simple guideline that limits the configuration space — such as hiding/removing unused parameters and highlighting commonly used ones — were proposed. Based on the simple guideline, the authors illustrated that in one of the studied projects, the configurations can be greatly reduced with minimal impact on the users side. Sayagh et al. [124] interviewed and surveyed developers on the design of run-time software configuration options. Accompanied by a detailed survey on the current literature on the subject. Their interview and survey suggests that additions of software configurations are often unplanned and by any developer. This practice can sometimes make configurations hard to maintain as the responsibility is unclear. Meanwhile, removing options can also be risky as it can break the software system unintentionally. A side effect by this is that software systems can have a high percentage of options that rarely change. Xu et al. [168] proposed a tool to support users through inferring configuration requirements automatically. Through mining constraints from the configuration space, their tool can expose misconfiguration vulnerabilities and identify error-prone configuration design and handling.

Unlike previous research which heavily focuses on configurations for production software, in this thesis we focus on *dotfiles* from the user space. Compared to production software, many configurations in the user space do not have high focus on performance or have a definitive approach to configuration. User-specific configurations are often tailored to the specific developers need and often change over time.

3.2.2 Developer Workflow

Developers' time and interest is a valuable resource. Many studies have focused on how to develop or improve tools to increase developer productivity. More importantly, studies have shown that tool choice does matter for developers [74]. And researchers have also found surprising ways (e.g., on the toilet) of how developers discover new tools [92].

Current research often focuses on one particular tool or a specific set of tools, yet do not look at the dynamics between the tools. For example, Schröder et al. [125] performed an

empirical study on aliases used during command-line customization. Their study suggests that developers mainly use aliases for shortcuts, modifications, and scripting. Johnson et al. [69] investigated the reasons why developers look away from existing static analysis tools hoping to gain insight on how to improve them. In another study, Damevski et al. [36] studied developer interactions in Visual Studio to detect potential usage problems. These efforts have the same goal to improve the developer efficiency, however choosing and adopting the best practices and tools is a challenging task for developers [132]. Some attempts have been performed to tackle this issue, for example, in a follow up study, Snipes et al. [133] explored the possibility to gamify the process and received dividing feedback from a pre-study survey.

Another heavily studied aspect of improving developer workflow is to provide improved code completion. Tools and new implementations emerge on the topic often; they are highly impactful and often causes excitement within the developer community. While these are not directly related to the current research, given the prevalence and high amount of activity related to text editors, we believe they could be relevant to the configuration of the text editors. Noticeable recent events includes the introduction of entire line completion by *tabnine* [142], language servers protocol by Microsoft [88] and GitHub Copilot [52]. Code completion research started from better recommendations from parsing source code and API information [65] to leveraging statistical models [113] and machine learning techniques [141] to provide better candidates. This feature can help developers write code more quickly instead of continually referencing API documentation [102]. Such a system often accompanies the basic AST parsing technique on IDEs and aims to provide a more accurate prediction to developers. However, a recent study by Hellendoorn et al. [62] on the real-word usage of completion have shown that despite many of the techniques works particularly well on synthetic datasets, their performance can drop to only 20% accuracy on hard tasks. In their study, they also find that most developers see the feature of code completion as saving the number of keystrokes typed and do not worry too much about the absolute accuracy of the prediction (i.e., the candidate showing up as first choice is not as important as the candidate showing up in the selection).

In this thesis, we focused on *dotfiles* management by developers. Since *dotfiles* is a collection of user-specific configurations, instead of focusing on a specific aspect and on a specific purpose, our study is a high-level view of how these configuration are managed and maintained over time.

3.3 App Stores in Software Engineering Research

Overall most research do not directly study app stores, but instead either study the applications offered in the app store or specific non-technical attributes of an app store. We go into detail discussing the differences and argue why in both cases the app store entity can have an affect on the associated research.

3.3.1 Overview of App Store Research

Since the idea of an app store have only been popularized in the mobile age, To date, research in this area has concentrated on a narrow set of app stores that primarily involves mobile platforms. In both of the earliest work focusing on app stores, they have selected a mobile app store as the investigation target. Harman et al. [61] proposed app stores as a valid kind of software repository worthy of formal study within the broader research area of mining software repositories; while their work was not specific to mobile app stores, they used BlackBerry App World as their canonical example. Ruiz et al. [121] studied the topic of reuse within app stores, focusing their work on Android Marketplace.[4] In addition to them both tacitly used only mobile app stores, in both cases, no formal definition of an "app store" was provided.

The heavy focus on the mobile scene, and the lack of treating stores as a proper platform, can be universally observed across all app store research as suggested by a survey in 2016 focusing app store research. In the survey, Martin et al. [85] observed that studies have often focused on only a few specific app stores, and have ignored comparisons between app stores. The same results can be suggested from a more recent survey focusing on app reviews, Dąbrowski et al. [35] found the median number of app stores studied to be 1, with the maximum being 3.

In practice, we can observe situations where similar app stores would have different impact on both users and developers. For example, in web extension stores, we can observe cases where the same extension is accepted in one store but rejected in another.[5] Sometimes, even the same features from different app stores can behave differently, for example, Lin et al. [80] found that reviews of games from Steam is dramatically different from reviews of apps in mobile app stores.

[4] Android Marketplace has since been re-branded as Google Play.

[5] https://github.com/FastForwardTeam/FastForward/issues/704

3.3.2 Involvement of App Stores in Software Engineering Research

To better understand app stores' involvement in software engineering research, we reviewed relevant papers from the two flagship software engineering research conferences: the *ACM/IEEE International Conference on Software Engineering* ("ICSE") and the *ACM SIGSOFT International Symposium on the Foundations of Software Engineering* ("FSE"). We used *Google Scholar* to find papers containing the keyword "app store" between January 2020 and April 2022 for the two conferences. We found a total of 34 such papers (listed in Appendix A). We note that our efforts do not constitute a comprehensive literature survey; instead, our goal was to gain an overview of how app stores are involved in recent research, and why app stores matter in their context.

» *Mining software applications* — App stores have been extensively used as a mining source of software applications. It can be observed in 20 of the 34 papers found. In these papers, the major focus is often on another subject and app stores provide a source where they can collect applications for either a data source or verification dataset. For example, Zhan et al. [177] proposed an approach to detect software vulnerabilities in third-party libraries of android applications. They leveraged the app store to collect a dataset to verify the effectiveness of their approach. In these studies, the app store is both a convenient and practical source of data collection. However, the involvement of app stores may not be necessary since the purpose is to gather a dataset of application. In Yang et al.'s work [169], they leveraged android applications from an existing dataset without the need to collect from an app store. We argue that the importance of app stores in these types of studies is the selection criteria performed by the researchers to collect applications from app stores. These features can include star ratings, total downloads, app category.

» *Mining app store artifacts* — In these studies, researchers focused on unique software artifacts that come from the operation of the app stores. App stores have a much heavier involvement in these studies compared to the previous. App reviews is the major software artifact the researchers focused on, where they leverage the data to identify features of applications [165], locating bug reports [59], and detect undesired app behaviors [66]. One interesting data we have observed is where van der Linden et al. [149] have leveraged the developer contact information shared on app stores to send out surveys related to security practices.

3.3.3 Store-focused research

As stated above, we found that most recent research involving app stores focuses on the applications they offer rather than on studying the app stores themselves; in particular, most research in the domain focuses on the development of mobile applications. Meanwhile, a few work have specifically considered app stores and their effects on software engineering, but again these works focus heavily on mobile app stores.

In a recent paper, Al-Subaihin et al. [3] interviewed developers about how app stores affect their software engineering tasks. They found that developers often leverage the review section from similar applications to help with understanding the expected user experience and anticipated features. App stores also provides a playground for releasing beta version of apps to receive feedback from users. The built-in communication channels also play a large role in informing development. The interviews suggest that developers pay attention to viewing user requests in app store via channels such as reviews and forums. The approval period of app stores affects how developers plan their release. App stores introduce non-technical challenges in the development process. Given the app store model of release, app store-specific metrics, such as total number of downloads, are considered highly important to developers.

Running an app store presents both technical and non-technical challenges to the store owner. Technical challenges include verifying that each app will install correctly, while non-technical challenges include ensuring that the promotional information in the app's product page adheres to store guidelines. Wang et al. [153] investigated several *Android* app stores in China and compared them to Google Play. Their study showed that these stores were much less diligent in screening the apps they offered, with a significantly higher presence of fake, cloned, and malicious apps than Google Play.

Chapter 4

Understanding Question Discussions on Stack Overflow

While Stack Overflow has become the largest knowledge base of Q&A content, it follows a question and answer centric view of Q&A. Prior research on Stack Overflow also often adopt a model of Q&A on the website as a two step process, where one user ask the question and the other user provides an answer directly. However, upon investigating Stack Overflow, we noticed a large amount of comments directly associated with questions on the platform. In this chapter, our primary goal is to understand the user interactions in question associated comments as question discussions.

Related publication An earlier version of the work described in this chapter has been published in the following paper:

> Wenhan Zhu, Haoxiang Zhang, Ahmed E. Hassan, Michael W. Godfrey. 2022. An empirical study of question discussions on Stack Overflow. Empirical Software Engineering.

4.1 Introduction

Stack Overflow is a technical question answering (Q&A) website widely used by developers to exchange programming-related knowledge through asking, discussing, and answering questions. The Q&A process on Stack Overflow creates a crowdsourced knowledge base that provides a means for developers across the globe to collectively build and improve their

knowledge on programming and its related technologies. Stack Overflow has become one of the largest public knowledge bases for developers with more than 21.9 million questions as of December 2021 [45]. A survey shows that retrieving information from Stack Overflow is an essential daily activity for many software developers [139].

On Stack Overflow, users can ask, answer, and discuss questions, and each question can receive multiple proposed answers. The user who asked the question (i.e., the "asker") can decide to mark one answer as *accepted*, indicating that it resolves their question authoritatively. While ultimately Q&A is the most important activity on Stack Overflow, users can also post comments and/or start chat rooms that are tied to a specific *post* (i.e., question or answer). In this chapter, we refer to comments and chat rooms messages on Stack Overflow as *discussions*; each discussion is associated with a single question (a *question discussion*) or proposed answer (an *answer discussion*).

Researchers have extensively studied the questions and answers on Stack Overflow. These studies ranged from finding out common question types [5] to predicting the best answers [144]. The Q&A processes on Stack Overflow are commonly viewed as two independent events by the studies. The first event is asking the question; this occurs when a user posts a question on the platform. The second event is answering the question; this normally occurs when another user posts an answer to a question. However, commenting as a communication channel allows for user interactions beyond simple asking and answering. A recent study has shown that comments can be helpful in the Q&A process by providing support knowledge, such as code examples, references, and alternative suggestions [126], whereas previous research has focused primarily on answer comments. Some studies leverage answer comments to study the quality of answers on Stack Overflow. For example, Zhang et al. [179] leveraged comments highlighting obsolete information regarding their associated answers. As a Q&A platform, most content on the platform is consumed by viewers long after the question is answered. If misleading information exists on the platform, it can convey false information within the community. Another study [28] used comments as a sign of whether the community is aware of the security vulnerabilities contained in the answer. Meanwhile, some studies have also focused on the presentation of knowledge on Stack Overflow. These studies also approach the issue from the answer perspective. One study [178] highlights that while users are reading answers on Stack Overflow, they should not ignore the information contained in their associated comments. In Zhang et al.'s next study [180], they showed that the current mechanisms on Stack Overflow to display comments is not ideal and can hurt the users when they are reading answers.

In this chapter, we focus on question comments. More specifically, we theorize that the commenting activities form a discussion and our focus is to understand how the discussions

affects the Q&A process on Stack Overflow. Unlike previous studies that mostly focus on answer comments which occur after a question has been answered, in this chapter we focus on question comments which can occur before the question is answered.

To help understand why it is important to study how question discussions integrate with the Q&A process, we now consider a motivating example. Fig. 4.1 shows a question titled *"Unable to set the* `NumberFormat` *property of the* `Range` *class."*[1]." Four minutes after the question was asked, another user posted a comment — attached to the question — asking for clarification on the problematic code snippet. A chat room was then created for the asker and the user to continue the discussion in real-time. A consensus was reached in the chat, and the results were summarized and posted as a proposed answer by the user, which the asker designated as *accepted*. This example highlights how the process of asking and answering questions is enabled by the discussion mechanisms of commenting and chatting, allowing a resolution to be reached quickly. That is, the question discussion can serve as a simple and effective socio-technical means to achieving closure on the question.

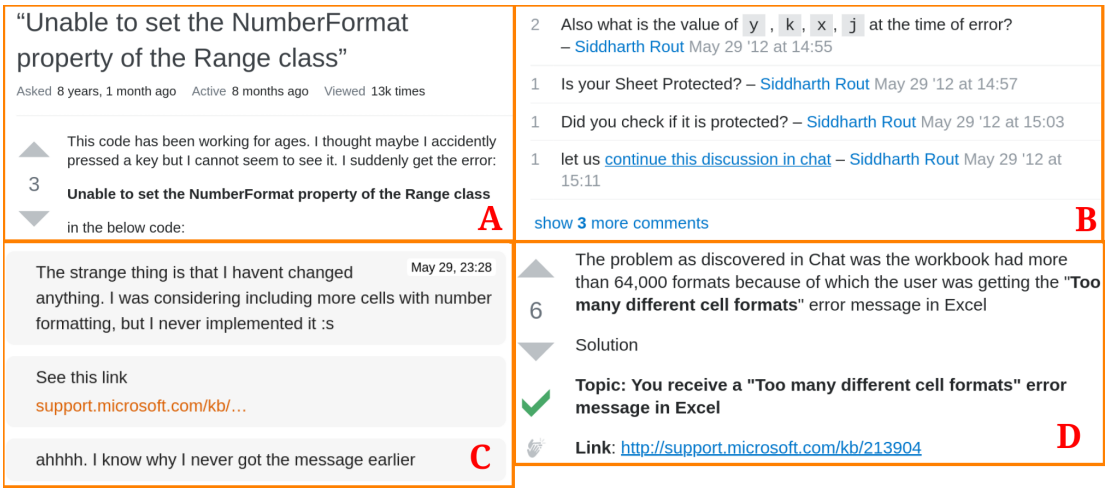


Figure 4.1: An example of the Q&A process involving discussions: (A) a user (the "asker") asked a question; (B) another user (the "answerer") started discussing with the asker in the comment thread; (C) the question was further clarified then resolved in the chat room; (D) the content of the comments and chat messages that led to the resolution of the question were summarized as an answer, which was marked as the accepted answer by the asker.

In this chapter, we use the Stack Overflow data dump from December 2021 [45] as our dataset; this dataset contains 43.6 million comments and 1.5 million chat messages. We

[1] https://stackoverflow.com/questions/10801537/

use this data to explore the nature of question discussions and how they integrate with the crowdsourced Q&A process on Stack Overflow. To make this study easy to follow, we use the following notations to refer to different groups of questions observed within the dataset:

Symbol	*Meaning*	*# in dataset*
Q_{disc}	Questions with comments	13.0 M
Q_{chat}	Questions with chat rooms (and comments)	27,146
Q_{nd}	Questions with no discussions	9.0 M
Q_{a}	Questions with answers	18.8 M
$Q_{d/a}$	Questions with both discussions and answers	10.5 M
$Q_{d/aa}$	Questions with both discussions and accepted answers	6.1 M
$Q_{hd/a}$	Questions with both discussions with "hidden comments"[2] and answers	1.6 M

Specifically, we investigate and answer three research questions (RQs):

RQ1: How prevalent are question discussions on Stack Overflow?

We found that question discussions occur in 59.2% of the questions on Stack Overflow. More specifically, 13.0 million questions have comments (i.e., Q_{disc}) with a median of 3 comments, and 27,146 questions have chat rooms (i.e., Q_{chat}). The popularity of question discussions is also increasing, with the proportion of questions with discussions nearly doubling from 32.3% in 2008 to 59.3% in 2018. Question discussions exist in all phases of the Q&A process on Stack Overflow. In questions that are both discussed and have an accepted answer (i.e., $Q_{d/aa}$), discussions in 80.6% of the questions begin before the accepted answer was posted. We found that the duration of question discussions can extend beyond the Q&A process: In 28.5% of $Q_{d/aa}$, question discussions begin before the first answer and continue after the accepted answer is posted; and in 19.4% of $Q_{d/aa}$, question discussions begin after the question receives its accepted answer.

RQ2: To what extent do users participate in question discussions?

We found that 16.0% (i.e., 2.6 million) of registered users on Stack Overflow have

[2]In Stack Overflow, comments are "hidden" (i.e., elided from view) by default when there are six or more attached to the same question.

participated in question discussions, which is comparable to the number of users who have answered questions (i.e., 16.7%). Question discussions allow askers and answerers to communicate with each other directly, enabling fast exchanges on the issues of concern. For questions that have both discussions and answers (i.e., $Q_{d/a}$), we found that as the number of comments increases, both askers and answerers were more likely to participate in the question discussions. Also, we found that when there are six or more comments present (i.e., $Q_{hd/a}$), then there is a high likelihood of both askers (90.3%) and answerers (51.9%) participating in the discussions.

RQ3: How do question discussions affect the question answering process on Stack Overflow?

Question discussions tend to lead to more substantial updates to the body of the original question. For example, a median of 114 characters are added to the question body when the question discussion has a chat room instance (i.e., Q_{chat}). While most other questions have no change in their question body length, a larger proportion of questions with comments are revised, with an increase in the question body length compared to questions with no discussion. Questions with more comments receive answers more slowly, with a Spearman correlation of $\rho = 0.709$ between the number of comments and the *answer-receiving-time* for the first answer.

The main contribution of this chapter is to highlight that discussions are an integral part of the Q&A process on Stack Overflow. Compared to the common assumptions that asking and answering questions are separate events in many studies, our work suggests that a large proportion of questions on Stack Overflow are answered after interactions between askers and answerers in question discussions. Our study suggests that question discussions is a very common activity comparable to answering activity on Stack Overflow. Question discussions have a high active user base (i.e., 16.0% of active users), and are also comparable to answering (i.e., 16.7% of active users). We also observed a strong correlation between the number of comments and the question answering speed, suggesting that question discussions have an impact on creating answers. Our findings suggest that question discussions can facilitate the Q&A process since they provide a means for askers and potential answerers to communicate throughout the Q&A process.

4.2 Data Collection

We use the Stack Overflow data dump from December 2021 [45]. The data dump is a snapshot of the underlying database used by Stack Overflow; it contains all meta-data for each comment, such as which user posted the comment and which question the comment is associated with. We mainly used the `Posts` and `Comments` table from the dataset to extract the required information. The data dump also contains the history of each question, via the `PostHistory` table. We analyze the history of each question to reconstruct the timeline of when the question was created, edited, commented, and answered.

Data about chat rooms is not contained in the Stack Overflow data dump; instead, we collected it manually by crawling the Stack Overflow website itself.[3] We also labeled the chat room instances based on whether they are *general*[4], attached to a *question*, or attached to an *answer* based on the structure of the crawled data. After cross-referencing their associated question IDs with the Stack Overflow data dump, we removed chat room discussions that are unrelated to programming, such as those on Meta Stack Overflow which focuses on the operation of Stack Overflow itself. This left us with a total of 27,312 chat rooms comprising 1.5 million messages that are associated with 27,146 questions as of December 2021. Fig. 4.2 shows the detailed extraction process of chat rooms from Stack Overflow.

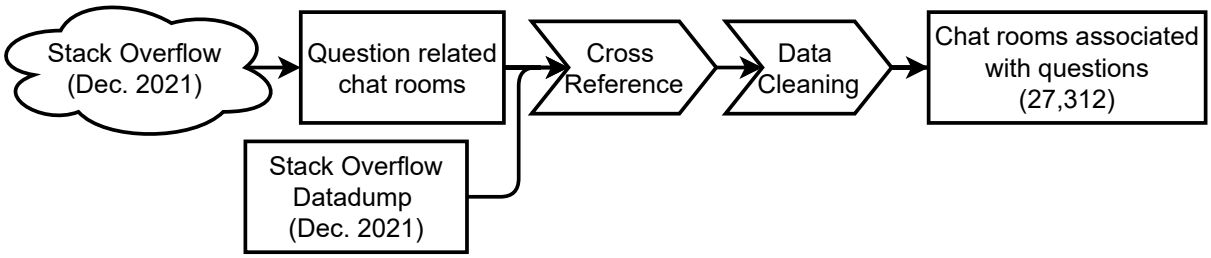


Figure 4.2: An overview for the creation of Q_{chat} (questions with chat rooms)

[3] We've made our dataset open access on Zenodo: https://zenodo.org/record/5516190

[4] General chat rooms are standard chat rooms on Stack Overflow that are not associated with a question or an answer.

4.3 Study Results

In this section, we explore the underlying motivation, the approach taken, and the results of our three research questions (RQs) concerning question discussions on Stack Overflow.

4.3.1 RQ1: How prevalent are question discussions on Stack Overflow?

Motivation: As a technical Q&A platform related to programming, Stack Overflow hosts a large number of questions [145]. From the user's point of view, creating an answer can be challenging since the initial version of a question is often incomplete or ambiguous. For this reason, potential answerers may first wish to engage the asker in a discussion to clarify their intent and possibly seek additional context, which is typically done using comments attached to the question. If the discussion proves to be fruitful, the user may then post an answer based on the discussion; also, the asker may decide to edit the original question to clarify the intent for other readers. Example 4.3.1 shows a comment pointing out a confounding issue in the original question. After the discussions, the asker acknowledged the issue and edited the original question for clarity.

A prior study showed that active tutoring through discussions in chat rooms can substantially improve the quality of newly posted questions by novice users [49]. However, it is labor intensive to provide such tutoring with an average of more than 7,000 new questions posted per day on Stack Overflow in 2019. At the same time, there has been no detailed study of question discussions as yet; in this RQ, we explicitly study question discussions to gain a better understanding of their prevalence in the Q&A process.

Example 1

In a comment linked to a question titled: "Write to Excel — Reading CSV with Pandas & Openpyxl - Python."[5], a user observed that the example CSV file given in the question did not follow the CSV standard, and suggested the asker to double check the input format.

Comment:

> *The structure of the first three lines doesn't match the structure of lines 5 onwards so you cannot read this file with a CSV library. Please check the provenance of the file and what it should look like. I suspect you probably want to skip the first four lines.*

Approach: We begin our study of the prevalence of question discussions by investigating the trend in the number and proportion of question discussions over the years. We distinguish between answered questions with and without an accepted answer to investigate whether there exists a difference between the two groups of questions.

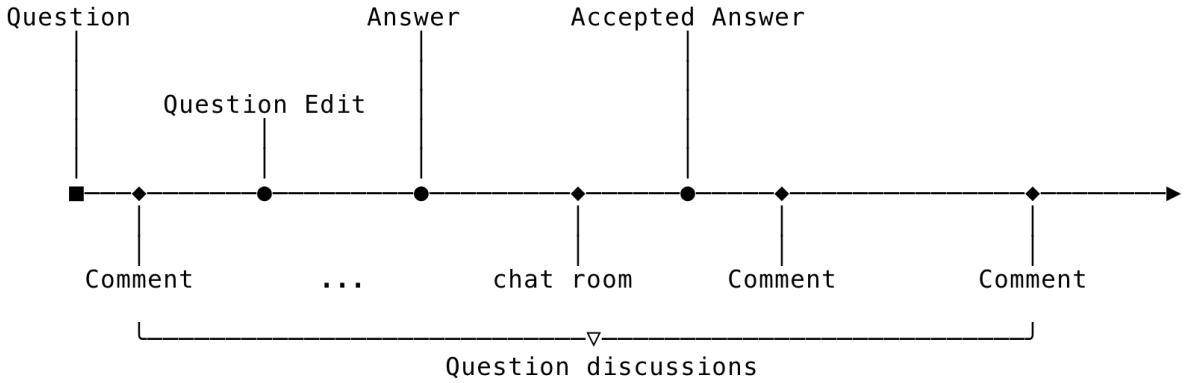


Figure 4.3: Timeline of question thread events. Question discussions can occur at any time since the creation of a question.

We then study when question discussions occur relative to key events in the Q&A process. After a question is posted on Stack Overflow, several different types of follow-up events may occur, as illustrated by Fig. 4.3. For example, after a question is posted any of the following can occur:

[5] https://stackoverflow.com/questions/48956597/

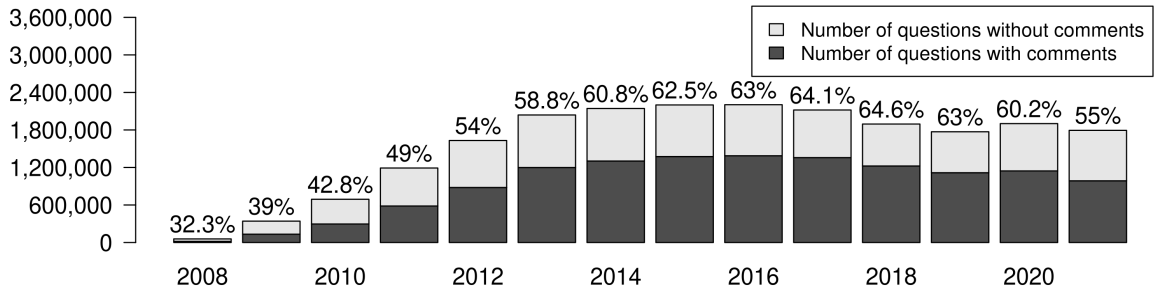
- other users can propose answers to the question;
- users can post comments to discuss either the question or the associated answers;
- the asker can mark one of the answers as *accepted*; and
- the question (and proposed answers) can be edited for clarity.

For each question, we construct the timeline consisting of each event, and we analyze the prevalence of question discussions with respect to other Q&A activities. Here, we focus mainly on two key events: when the question receives its first answer, and when it receives the accepted answer.

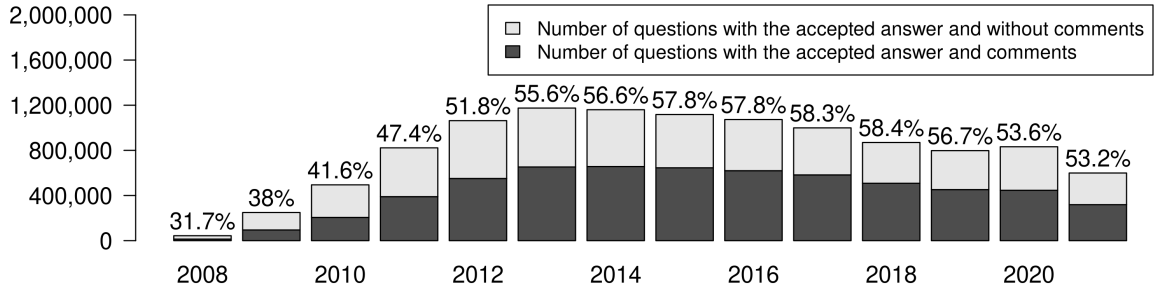
Results: Stack Overflow questions are discussed by 43.6 million comments and 1.5 million chat messages, forming a large dataset of community question discussions, in addition to the 22.0 million questions and 32.7 million answers. The proportion of questions with discussions also nearly doubled from 32.3% in 2008 to 59.3% in 2013, and has remained roughly stable since then. Fig. 4.4a shows the number and proportion of questions with discussions per year, and Fig. 4.4b suggests a similar trend for questions with an accepted answer. Since a question may receive its first comment several years later, it is likely that the proportion of recent years will increase slightly in the future.

Question discussions occur throughout the Q&A process, ranging from before the first answering event to after the accepted answer is posted. Fig. 4.5 shows the proportion of question discussions relative to answering events in the Q&A process. The height of the band across each vertical line indicates the proportion of questions with a specific activity occurring in that phases of a question thread's life cycle. For example, from the left-most bar, all questions can be split into two groups: questions with discussions (Q_{disc}) and questions without discussions (Q_{nd}). The top band (with strata in blue) represents 59.2% of the questions with discussions and the bottom band (with strata in red) represents 40.8% of the questions without any discussions. Flowing from left to right, the strata in blue and red continue to represent the questions with and without discussions until the right most band where it represent the final answering status of the question.

In $Q_{d/a}$, 76.2% (i.e., 8.0 million) of the question discussions begin before the first answer is posted, suggesting an influence of question discussions on answering activities. Furthermore, 80.6% (i.e., 4.9 million) of the question discussions begin before the accepted answer is posted, indicating a slightly more active involvement



(a) All questions



(b) Questions with the accepted answer

Figure 4.4: The number and proportion of questions with comments

of question discussions in $Q_{d/aa}$. In answered and solved questions of Q_{chat}, 59.1% (i.e., 12,507) of the chat activities begin before the first answer is received, and 72.9% (i.e., 10,172) of the chat activities begin before the accepted answer is posted.

The early occurrence of question discussions in the Q&A process suggests that they enable interested users to engage with the asker informally, to allow for clarification. In Example 4.3.1, 13 minutes after the question was initially posted, a user asked for a concrete example that can demonstrate the problem the asker had. The asker then updated the question with the requested information. The question was answered 15 minutes later, incorporating the newly added information based on the discussions.

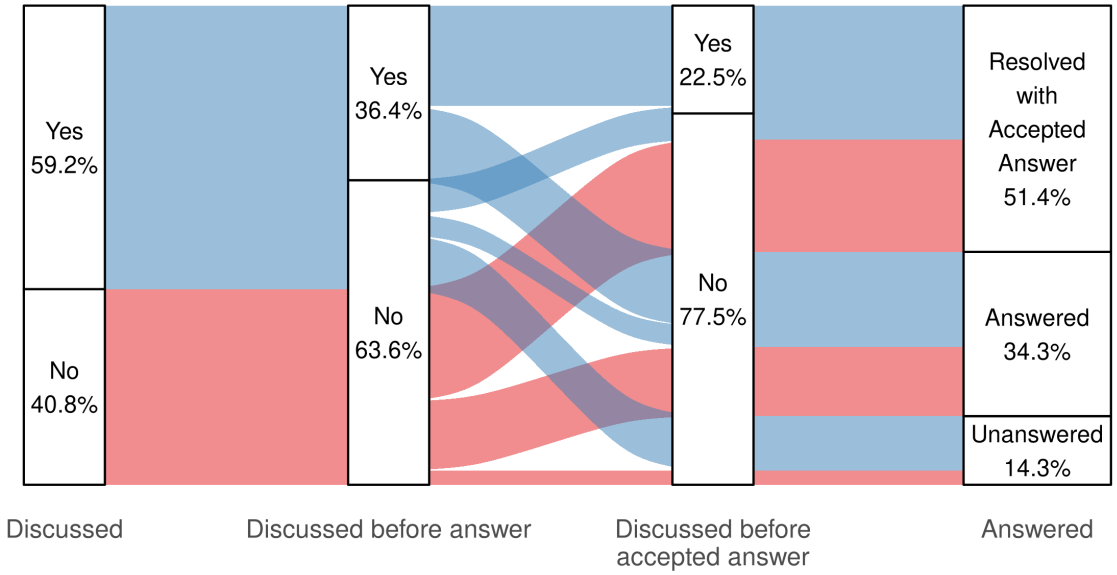


Figure 4.5: Question discussion with respect to answering events during the Q&A process. The blue bands represent questions with discussions and the red bands represent questions without discussions.

Example 2

A user comments to ask for information in a question titled "Can I modify the text within a beautiful soup tag without converting it into a string?"[6]

Comment:

> *UserB: Please give an example html that demonstrates the problem. Thanks.* [2014-09-16 13:15]
> *UserA (the asker): Just added some example html, sorry about that.* [2014-09-16 13:20]

In 28.5% (i.e., 1.7 million) of $Q_{d/aa}$, the discussions begin before the accepted answer has been received, and continue after the accepted answer is posted. Furthermore, 19.4% (i.e., 1.2 million) of the question discussions begin after the accepted answer is posted. These findings indicate that the community may

[6] https://stackoverflow.com/questions/25869533/

continue to discuss questions even after the asker has designated a "best" answer that solves their problem [9]. This may be due to the fact that software development technologies tend to evolve rapidly; old "truths" may need to be updated over time, and additional discussions may provide new insights despite the asker considering the question to be solved. Example 4.3.1 shows a comment that pointed out a potential security vulnerability in the code snippet 5 years after the initial question is posted.

Example 3

A user posted a comment to warn about a potential security vulnerability 5 years after a question was posted."[7]

Comment:

> *Beware. If you've configured your Struts application in this particular way (setting 'alwaysSelectFullNamespace' to 'true'), your application is very likely vulnerable to CVE-2018-11776:* semmle.com/news/apache-struts-CVE-2018-11776

RQ1 Summary:

There are 44.6 million comments and 1.5 million chat room messages in our dataset, which forms a large corpus of question discussion activities on Stack Overflow. Since the introduction of comments, the popularity of question discussions has nearly doubled from 32.3% in 2008 to 59.3% in 2013 and has remained stable since. The occurrence of question discussions is prevalent throughout the Q&A process. While question discussions in most questions (76.2% in $Q_{d/a}$ and 80.6% in $Q_{d/aa}$) begin before the answering activities, question discussions can continue or even begin after the accepted answer is posted.

[7] https://stackoverflow.com/questions/17690956/

4.3.2 RQ2: To what extent do users participate in question discussions?

Motivation: The crowdsourced Q&A process on Stack Overflow is driven by user participation. In addition to the questions and answers, question discussions are also part of the user-contributed content on Stack Overflow. In this RQ, we explore how different users participate in question discussions, to better understand how question discussions facilitate the Q&A process.

We focus on two aspects of user participation. First, we investigate the overall user participation in question discussions on Stack Overflow. We note that in RQ1, we observed a high proportion of questions with discussions; here, we focus on the users who participate in question discussions. Second, we change the scope to focus on the question-level discussion participation. We are interested in what other activities that the participating users join in on. For example, did the user ask the question in the first place, or did the user post an answer for the question.

Approach: To study user participation in question discussions and gain an overall idea of the popularity of discussion activities compared to other activities on Stack Overflow, we extract from the data dump the list of all users who contributed content to Stack Overflow. In particular, we sought users who asked, answered, or discussed questions; we note that while other activities, such as voting, may help the community, we do not consider these activities in our study as they do not directly contribute content. We also ignored activity related to answer discussions, as it was outside of the scope of our investigations.

We extracted the unique *UserIDs* from all questions, answers, and question comments to build the groups of users who participated in each of those activities. We then compared the intersection between the different sets of users to determine which of them participated in multiple types of activities on Stack Overflow.

Results: 2.6 million (i.e., 16.0%) users on Stack Overflow have participated in question discussions. Fig. 4.6 shows the overlap of the number of users participating in different activities on Stack Overflow. We observe that 95.7% of users who participated in question discussions also asked questions on Stack Overflow, and 50.9% of them answered questions.

In 60.0% of $Q_{d/a}$ (i.e., 7.8 million), askers participate in the question discussions and in 34.1% of $Q_{d/a}$ (i.e., 3.6 million), an answerer participated in

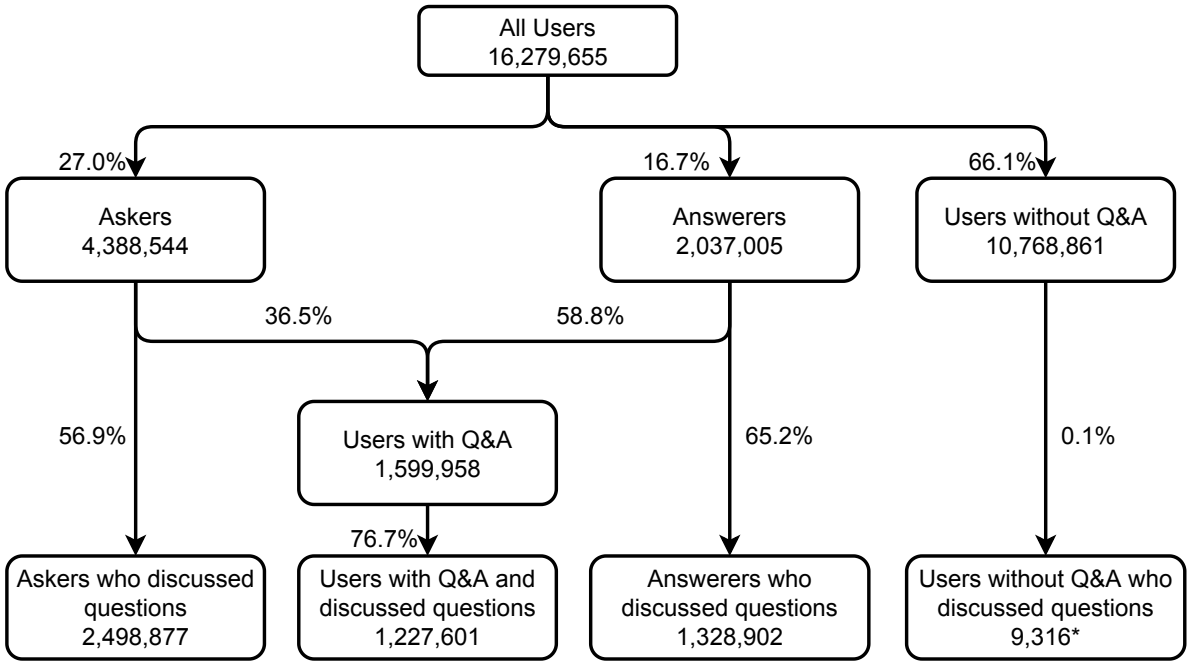


Figure 4.6: The number of users who participate in different types of activities on Stack Overflow, and the number and proportion of users who participate in question discussions.

the question discussion. The involvement of askers and answerers suggest that the two parties often leverage question discussions as a collaboration medium.

We further investigate the trend of the proportion of questions with askers and answerers in question discussions as the number of comments increases. **When the number of comments increases, a higher proportion of questions have askers and answerers participating**. Fig. 4.7 shows the trend of the proportion of askers and answerers participating in question discussions as the number of comments increases. When there are at least 6 comments associated with a question (i.e., when Stack Overflow starts to hide additional comments), askers are present in at least 90.3% of the question discussions and answerers are present in at least 51.9% of the question discussions. Moreover, **when answerers are present in a question discussion, 79.3% (i.e., 2.8 million) of the answerers and 81.1% (i.e., 1.5 million) of the accepted answerers joined the question's discussions before posting the answers.** The increasing proportion and early engagements of answerers in question discussions suggest that users are actively

leveraging the question discussions as a communication channel to facilitate the answering of questions.

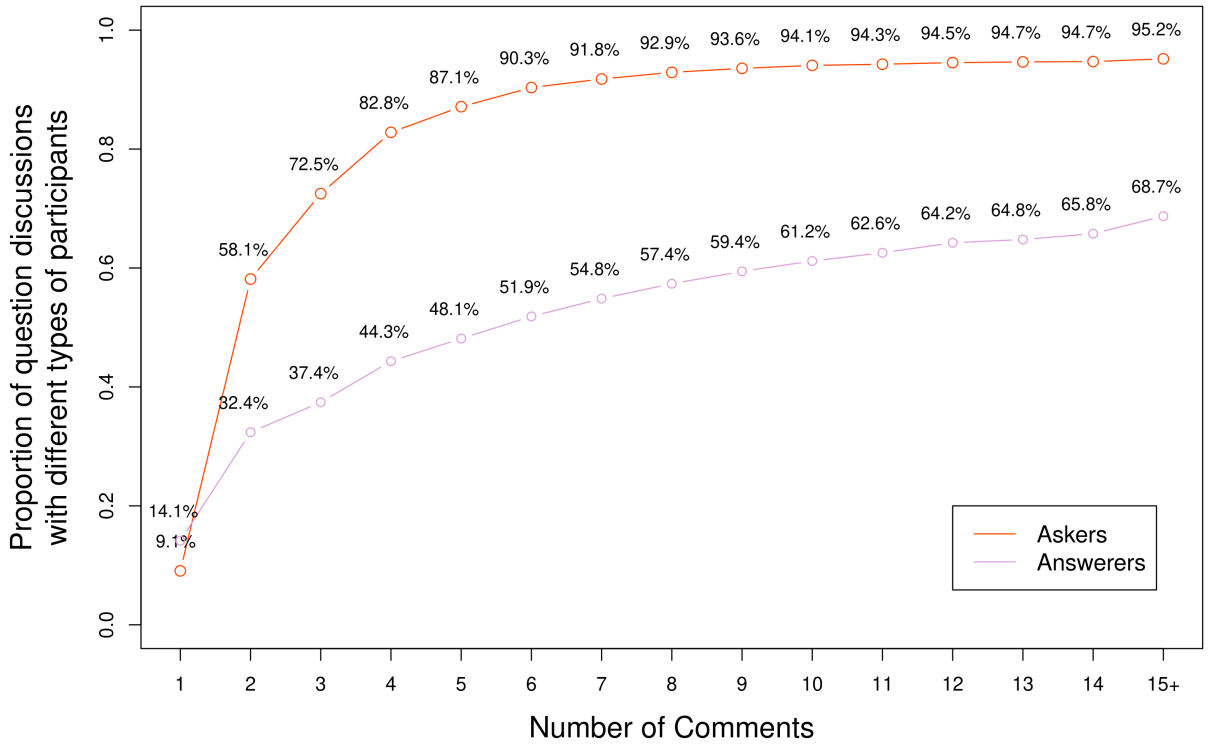


Figure 4.7: The proportion of question discussions with the participation of askers and answerers

RQ2 Summary:

2.6 million (i.e., 16.0%) users on Stack Overflow have participated in question discussions. These users overlap heavily with users who asked and answered questions on Stack Overflow. In $Q_{d/a}$, 60.0% of the questions have the asker participating in the question discussion and 34.1% of the questions have an answerer participating in the question discussion. The proportion of questions with askers and answerers participating in question discussions increases as the number of comments increases. When at least 6 comments are present, more than 90.3% of the discussions have askers participating and more than 51.9% have answerers participating. In 79.3% of $Q_{d/a}$ (81.1% of $Q_{d/aa}$), the answerer (accepted answerer) participated in the question discussion before they posted the answer (accepted answer).

4.3.3 RQ3: How do question discussions affect the question answering process on Stack Overflow?

Motivation: On Stack Overflow, questions serve as a starting point for curating crowdsourced knowledge. Devising a good question can also be a challenging task [22]. To encourage users to ask high-quality questions, in late 2019 Stack Overflow modified its reputation system to reward more reputation points on upvotes for questions, increasing the points rewarded from 5 to 10.[8] As noted previously, a question can have several follow-up answers; also, discussions can be associated with either the question or its answers. Questions (and answers) may be edited and revised by their original author, and this happens commonly.[9] This may be done to reflect new knowledge learned though the Q&A process, and to improve the quality of the posts themselves. In practice, some revisions are editorial or presentational in nature, such as fixing typos and formatting content for readability; however, questions are also edited to improve the quality of the crowdsourced knowledge [68]. Baltes et al. [14] observed that comments have a closer temporal relationship with edits than posts (i.e., a question or an answer), that is, the time difference between comments and post edits are smaller compared to comments and post creations. Typically, this happens for clarification purposes as answers and discussions shed new light on the original problem. For example, sometimes the asker's question may not include enough technical detail to be easily answered; similarly, the asker may conflate several issues into one posting. In these cases, the asker may seek to clarify the content of their

[8] https://stackoverflow.blog/2019/11/13/were-rewarding-the-question-askers/

[9] Comments may be deleted by their author, but they may not be edited in place.

question by adding new context or editing out extraneous details. Also, sometimes new answers emerge to older questions as the accompanying technologies evolve. Thus, it is important to recognize that the question discussions can affect the evolution of the question itself; the question version that appears to a casual reader may have evolved since its original posting.

In this RQ, we study how question discussions are associated with the evolution of questions. More specifically, we study the association between the number of comments and question revisions; we do so to better understand how question discussions affect the evolution of the question content. We also study the association between the number of comments and the *answer-receiving-time* to explore how question discussions affect the Q&A process.

Approach: To understand how question discussions affect the evolution of questions, we first study the correlation between question discussions and question revisions. Here, we are mainly interested in the scale of question edits in terms of the size of question content change in the question body. Specifically, we calculate the change in the number of characters in the question body between its initial version and the current version. We also categorize all questions into three groups, i.e., questions with no discussions (Q_{nd}), questions with comments (Q_{disc}), and questions with chat rooms (Q_{chat}). For each question from any category, we calculate the character length difference between the current version of the question and its initial version to investigate how question discussions are associated with the changes in the question content over a question's lifetime.

To understand how question discussions associate with the speed of question answering, we study the correlation between the number of received comments before answering activities and the *answer-receiving-time*. Similar to RQ1, here we investigate the *answer-receiving-time* of two different answering events: the *answer-receiving-time* for the first answer (i.e., t_{FA}) and the *answer-receiving-time* for the accepted answer (i.e., t_{AA}). For each question, we compute both t_{FA} and t_{AA}. We then group the questions by the number of received comments before the first answer and accepted answer respectively. Finally, we measure the Spearman correlation [137] between the number of comments and the median t_{FA} (t_{AA}) for questions with the same number of received comments before the first answer (accepted answer) is posted.

Results: Questions with chat rooms are more likely to be revised than questions without chat rooms, with a median size increase of 114 characters. Questions without chat rooms, on the other hand, do not exhibit a net change in size, although such questions may still receive edits. Thus, the existence of a chat room attached to a

question makes it more likely that the question will undergo significant revision. Fig. 4.8 shows the distribution of questions by the change in question body length after the question is posted, according to different levels of question discussion activities. From the figure, we can observe that while Q_{nd} and Q_{chat} share the same median and modal of zero characters change in question body length, a higher proportion of questions with comments receive revisions that lead to an increase in the question body length.

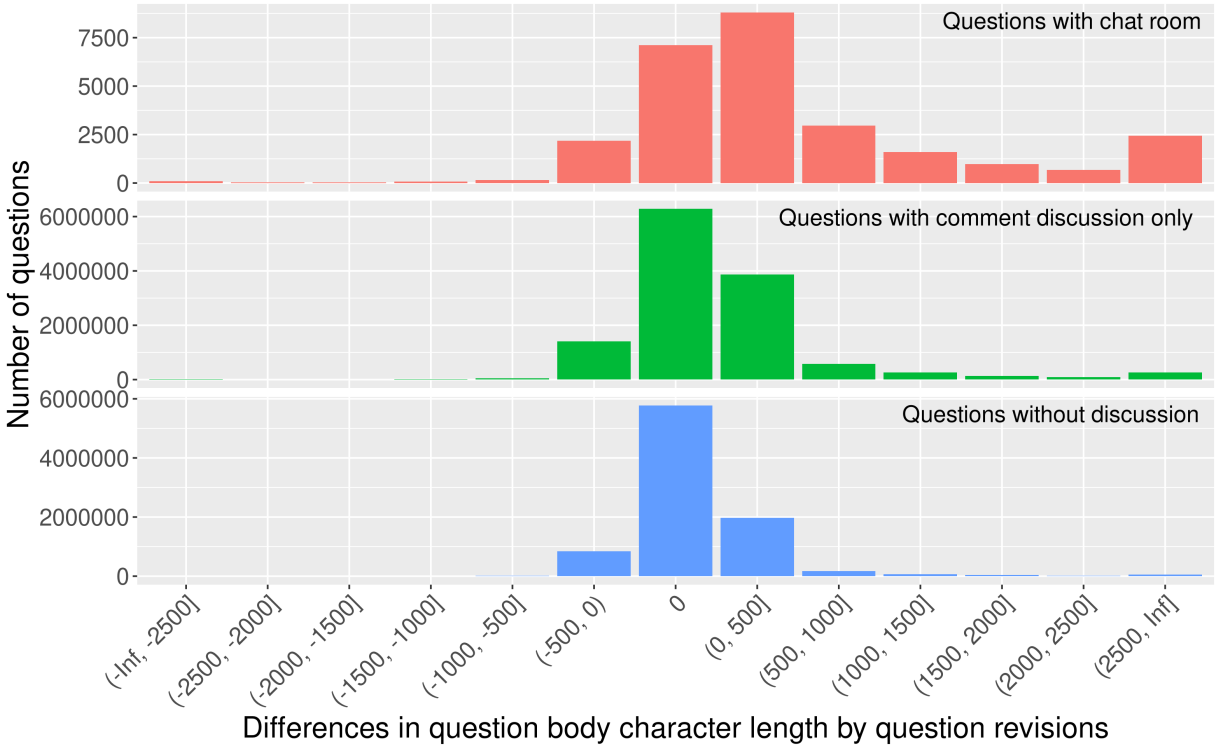


Figure 4.8: The distribution of the number of questions to the change in question body character length after the question is posted at different levels of question discussion activity

Overall, **the number of comments is strongly correlated with both t_{FA} (i.e., $\rho = 0.709$, $p \ll 0.05$) and t_{AA} (i.e., $\rho = 0.806$, $p \ll 0.05$)**. Fig. 4.9 shows the median t_{FA} and t_{AA} of questions with respect to the number of received comments before their respected answering events. Questions with many discussions also take a longer time to answer. One possibility is that the difficulty of these questions is also higher, therefore requiring more effort by the users to have an extended discussion before the question can be answered. At the same time, for the *answer-receiving-time* of Q_{chat}, we find that it takes a median of 5,935 secs (i.e., 1.6 hrs) and 8,438.5 secs (i.e., 2.3 hrs) to receive the first answer and

the accepted answer. The answering time follows the same trend of more discussions, i.e., a longer answering time. The strong correlation between the number of comments that a question receives and the *answer-receiving-time* suggests a close relationship between question discussions and creating answers. Our findings suggest that after a question is asked, interested users may offer help first in comments when an answer can't be created immediately. Therefore, they begin the Q&A process by discussing with the asker through commenting. This is also supported by our observations in RQ1 and RQ2 where discussions mainly begin before answering and a high proportion of answerers participate in question discussions.

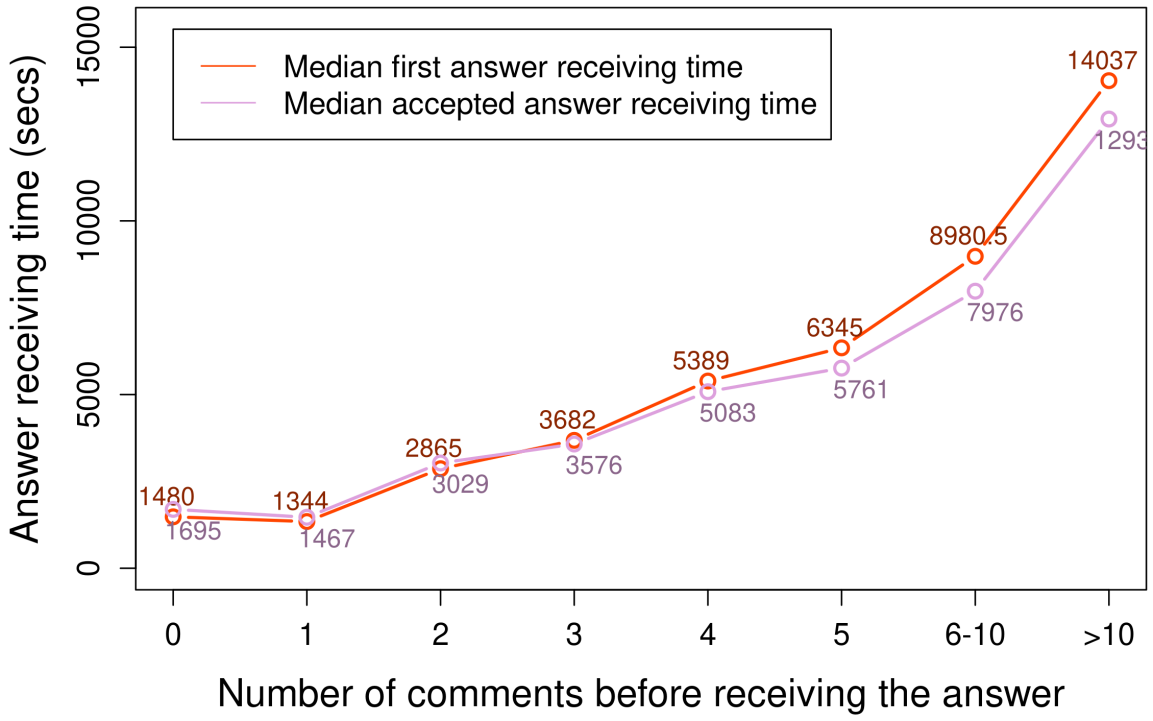


Figure 4.9: Median *answer-receiving-time* with respect to the number of comments that are posted before the answer. The median is only calculated for questions with answers and questions with accepted answers respectively.

RQ3 Summary:

Question revisions for Q_{chat} are more likely to lead to larger edits in the question body, with a median increase of 114 characters to the question body. A strong correlation exists between the *answer-receiving-time* and number of comments before the answer, suggesting its close relationship with answering activities.

4.4 Implications and Discussions

4.4.1 Feedback From the Community

We shared our empirical observations on Meta Stack Overflow[10], where users share ideas and discuss the operation of Stack Overflow. We are glad that the users from the community find our observations align with their personal experiences with question discussions on Stack Overflow.

Some users also shared their personal experiences of leveraging question discussions. For example, one user stated "Many questions are very localized (i.e. help only the OP (Original Poster)) and very easy to answer (i.e. obvious to anyone who has any experience). For these, writing a proper answer, with explanations, seems like a waste of time." It supports our theory that question discussions provide a means for alternative response than an answer. For questions with no answers, users may still find enough information in the question discussion that can be helpful.

Other users also noticed that question discussions may be a sign of new users not realizing the edit feature for questions, "One thing I've noticed is that new users don't seem to realize they can edit their questions. When asked for clarity, they often (attempt to) dump great chunks of information in a comment." The observation is supported by another user who stated "I always add a suggestion to [edit] the question unless I am sure the user knows how to do it. Such a suggestion is not offensive, and provides the user a convenient button to edit." These observations also aligns with our findings that discussed questions are often edited more in RQ3.

Some users observed that comments can be deleted on Stack Overflow; future studies may wish to investigate this practice. Since Stack Overflow data dumps capture only a static copy of the website, researchers could monitor newly posted questions in real-time to capture deleted comments.

[10] https://meta.stackoverflow.com/questions/416059/

Another observation the community raised is that "easy questions are often answered in comments". Users indicate that they find writing a quick comment can often help the asker quickly. However, this also introduces noise to the platform, and the reader may be uncertain where to look for such information.

4.4.2 Suggestions for Researchers

While Stack Overflow is the dominating platform for Q&A activities for developers, Q&A also exists in other platforms and often in other forms. Future research can focus on the differences between Q&A platforms to better understand the developer's preferences when asking questions. A better understanding of developer's Q&A needs can help us build better platforms and tools to preserve the information from the Q&A sessions across platforms and improve the knowledge retrieval of the information for future users.

Include discussions when modeling Stack Overflow. Many current studies have considered asking and answering questions as isolated events. After a question is posted, other users will read the question and try to answer it. However, our study suggests a different story for many questions. Discussions in the form of comments occur at large scale for questions on Stack Overflow. The prevalence of question discussions with askers and answerers participating significantly in them suggests that they play a key role in the overall Q&A process; consequently, any empirical study of the Stack Overflow Q&A process has much to gain by explicitly considering question discussions in their modeling. For example, many tools have been proposed by researchers to support developers by leveraging Stack Overflow as a knowledge base [21, 147, 176]. While, these tools mined the content of questions and answers to retrieve relevant information for developers, they do not leverage the information that is contained in question discussions. By considering question discussion in their modeling, we believe the effectiveness of these tools can be further improved with more information.

Design automated tools to highlight information in question discussions. Stack Overflow's overwhelming success with the international software development community is due largely to the high quality of its content, in the form of questions and answers with accompanying discussions. However, maintaining the quality and relevance of such a large knowledge base is a challenging task; a recent study found that low quality posts hurt the reputation of Stack Overflow [110]. Because programming technologies evolve quickly, the detailed information in the questions and answers can become obsolete [179] and requires continual updating. Therefore maintaining a high quality and up to date knowledge base is very important for its users. For this reason, Stack Overflow allows users to edit

questions and answers even after a clear consensus has arisen. Stack Overflow, as a major source of information for developers, currently does not have any mechanisms that are dedicated to the maintenance of knowledge on the platform. Since knowledge maintenance is essential to the community, our study shows that users leverage question discussion to aid the maintenance of knowledge in the question content. Previous studies have also observed similar phenomena in answers [179, 135]. We suggest future research to focus on the evolution of knowledge on Stack Overflow via commenting behavior to extract best practices of the process. By understanding the evolution of knowledge content on Stack Overflow, we can design better mechanisms on the platform to better support the community effort in maintaining knowledge. For example, there could be report buttons for questions and answers that can raise flags regarding false information, legacy information, or potential security flaws. Questions with such flags can be then examined by other users and therefore maintaining a knowledge base that is up to date.

4.4.3 Suggestions for Q&A Platform Designers

Stack Overflow uses a gamification system based on reputation and badges to reward users who participate in the Q&A process; for example, upvotes on questions and answers reward the original poster with reputation points. However, at present upvotes for comments do not boost the reputation of the commenter, so their system does not currently reward participation in discussions.[11] Since so much effort is put into discussions — as evidenced by the presence of 43.6 million comments and 1.5 million chat messages in the 2021 data dump — this seems like a missed opportunity. Stack Overflow could reward those users who, through their participation in discussions, help to clarify, explore, and otherwise improve the questions and answers themselves; our studies here have shown just how influential question discussions can be on improving the quality of the questions and answers. Rewarding participation in discussions would create a positive feedback loop in the Stack Overflow gamification system, which would in turn encourage more users to engage in discussions.

Acknowledge discussions as essential in the Q&A, and design systems that incorporate the users' need for discussions. A good piece of shareable knowledge starts with a good question, and Stack Overflow has practices to help ensure high quality questions. For example, when novice users (i.e., users with newly registered accounts) first ask questions, they are led through an interactive guide on how to ask a good question. The guide includes both conventions (e.g., *tag the question*) and best practices for asking

[11] https://meta.stackexchange.com/questions/17364/

questions (e.g., *include what has been attempted to solve the question*). Although Stack Overflow already has a detailed walk-through on how to ask a good question, we observed that in practice, discussing and revising questions remains commonplace. At the same time, crowdsourced Q&A is a labor intensive process; for example, a question may take time to attract the "right" answerers or a question may be hard to understand without clarification. In exploring RQ3, we observed that questions with extended discussions — especially those that continue into a chat room — tend to receive more edits to the question body. We conjecture that question discussions can serve as a feedback loop for the asker, resulting in improvements to the questions through subsequent edits. Our observation also echoes a previous study which shows that tutoring novice users before they post their questions can improve the quality of their question [49]. We wonder if a question quality assurance "bot" might be able to leverage the question discussion data and mining the discussion patterns to further support askers in efficiently getting answers through crowdsourced Q&A.

Offer real-time Q&A for urgent question asking, and encourage users to organize the information for future reading. Question discussions offer a means for askers and answerers to communicate with each other during the Q&A process. Currently, chat rooms are triggered automatically once three back-and-forth comments occur between two users. However, there are cases where two users may wish to start a live conversation immediately. For example, traditionally in the open source community, it is suggested to ask urgent questions in an IRC channel to receive an immediate response [115]. However, when users do so, the information during the Q&A session will be buried in the IRC chat log. On the other hand, if a user were to ask the question on Stack Overflow, in exchange for not having an instant response, the Q&A information will remain easily accessible by the public. While Stack Overflow already offers chat rooms as a means for instant and real-time communication, currently the chat room triggering mechanism in posting comments is an inefficient communication channel for such need. There exists a potential for users to choose between a synchronous or asynchronous discussion through chat rooms or comments, respectively. For example, Stack Overflow could build in a feature that allows users to indicate if they are available online, and are waiting for an answer. When other users see the indicator, they could directly start discussions in chat rooms, and later update the content of the question based on the discussion. An intelligent communication channel selection bot could be designed to help users seek an effective type of communication by mining the historical data of communication preferences. Furthermore, a content summarization tool could be designed to extract pertinent information from both comments and chat rooms, for future users to better understand the context of the evolution of a question.

4.5 Threats to Validity

» *External Validity* — Threats to external validity relate to the generalizability of our findings. In our study, we focus on question discussions on technical Q&A on Stack Overflow, which is the largest and most popular Q&A platform for programming related questions. As a result our results may not generalize to other Q&A platforms (e.g., CodeProject[12] and Coderanch[13]). To mitigate this threat, future work can consider studying more Q&A platforms.

Another threat is that the studied Stack Overflow data dump only the current copy of Stack Overflow's website data. For example, users are allowed to delete their comments, answers, and questions. This means that when users delete their comments, they are expunged from the dataset, and we are unaware of how those comments might have affected the rest of the discussion. This concern is also shared by community members as one user stated "the majority of the comments ever posted on Stack Overflow are probably deleted." Meanwhile, since there is always a valid reason for a comment to be removed, another users suggested that "it's actually good that deleted comments are not public and Stack Overflow data dumps only capture the snapshot at the time it was taken. We don't want this kind of comments (i.e., rude/abusive comments[14]) to linger for more than a quarter..." Since Stack Overflow releases their data dump quarterly, we perform a comparison between the data dump from Dec. 2019 and the data dump from Dec. 2021. From the 32.9 million question comments in 2019, only 2.1% (i.e., 689,476) comments have been deleted in the newer data dump. So in other words, we are unable to monitor comments that were posted and deleted within the releases of two data dumps. But if the comment survived initially, it'll likely last.

» *Internal Validity* — Threats to interval validity relate to experimental errors and bias. Our analysis is based on the data dump of Stack Overflow from December 2021 (the comment dataset) and web crawling in December 2021 (the chat room dataset). While the difference between the data dump and chat room crawling is only a month, Stack Overflow as a dynamic platform is subject to change and the data itself can evolve. Future work can assess our observations on new data and evaluate whether our findings continue to hold over time.

» *Construct Validity* — Since the Stack Overflow data dump not include chat room-related data, we mined that data directly from the Stack Overflow website. This means that our

[12] https://www.codeproject.com/
[13] https://coderanch.com/
[14] https://meta.stackoverflow.com/questions/326494/

crawler and the collected data may be subject to errors (e.g., crawler timeout). We mitigate this issue by manually checking a subset of the collected data and verified the correctness of the scripts.

4.6 Conclusions

Question discussions are an integral part of the Q&A process on Stack Overflow, serving as an auxiliary communication channel for many developers whose technical information needs are not fully met within their nominal work environment. Question discussions occur throughout all phases of the Q&A process, especially before questions are answered. In 76.2% of $Q_{d/a}$ and 80.6% of $Q_{d/aa}$, the question discussions begin before the first answer and the accepted answer is posted; furthermore, 19.4% of the question discussions begin even after the accepted answer is posted. Question discussions allow askers and potential answerers to interact and solve the question before posting an answer. In $Q_{d/a}$, askers participate in 60.0% (i.e., 7.8 million) of the questions discussions and answerers participate in 34.1% (i.e., 3.6 million) of question discussions. When the number of comments increases, a higher proportion of questions are participated by askers and answerers. The *answer-receiving-time* of a question is strongly correlated (i.e., with a Spearman correlation of $\rho = 0.709$) with the number of comments a question receives before its first answer. We believe that our study of question discussions can be leveraged in several ways to improve the Q&A process. For example, an automated triaging system could suggest an appropriate communication channel; also, bots could be designed to warn about questions that seem unclear and might require further clarification.

Chapter 5

Understanding the Practice of Maintaining User-Specific Configuration Files

Software developers leverage tools in the software development process to increase their productivity. Developers often customize the tools they use to suit their preference. The customizations are often stored in hidden plain-text files following the UNIX tradition. As a convention, these files are often referred to as *dotfiles*. There has been a lack of research in understanding how developers manage their *dotfiles*. Some studies have looked at specific customizations for specific tools, such as for shell aliases and bash scripts. In this chapter, we perform an empirical study to understand the practices of maintaining user-specific configuration files through the analysis of publicly shared *dotfiles* repositories.

Related publication An earlier version of the work is currently in the process of submission.

> Wenhan Zhu, Michael W. Godfrey. An empirical study on dotfile repositories containing user-specific configuration files

5.1 Introduction

Tools are essential in software development. Studies have shown that familiarity with tools can significantly increase developer productivity [70]. Given the complexity and

diversity of software development tasks, software tools are often designed to be highly versatile, with a huge selection of configuration options; some also support scripting, which enables developers to construct complex customized usage scenarios. Furthermore, some configuration options are based purely on user preferences (e.g., shell aliases, editor color scheme), thus no optimal configuration exists for all usage scenarios. As a convention, user-configuration files are often referred to as *dotfiles*. Detailed information regarding the history of the name *dotfile* can be find in Section 2.2.

To illustrate the importance of *dotfiles* in software development, we present a motivating example of configuring "hotkey" commenting/decommenting in the text editor *Vim*. Fig. 5.1 presents a simple configuration that defines two "hotkeys" to allow *Vim* users to add or remove *C*-style comments in source code with a single key-press.

```
1 nnoremap <F2> :norm ^i// <C-[>
2 nnoremap <F3> :norm ^3x<C-[>
```

Figure 5.1: Simple configuration for toggling comments in *Vim*

While this short configuration is functional, there is room for customization. When the configuration is publicly available, through *GitHub* or a community wiki, other users may leverage the configuration as a base point and adapt the configuration for improvement or personal use. For example, another user may wish to change the configuration to use different hotkeys that they prefer; or, they may improve the script to support commenting in additional programming languages (e.g., `#` in *Bash*). If these changes are shared back to the community, the author of the original configuration may adapt their own script to add the improvements.

This process of sharing *dotfiles* is similar to what Eric Raymond described as the bazaar type [114] of open source development. In current days, most users of *Vim* will likely use a plugin manager, and leverage a plugin to take care of commenting functionality. We want to highlight that the customizability of software tools enable users to adapt the tool to their own use case. Meanwhile, with a collection of user customizations, we can understand the usage of the tools in real-world scenarios and in turn help improve the tools. For example, it is now very hard to find a text editor without the feature of hotkey commenting already built-in. We believe this is due to later tools learning from earlier tools in user customizations.

Although there is a vast body of empirical research on how software developers use their tools [69, 36, 133, 92, 74], at the same time there has been little study of how developer

manage their *dotfiles* over the long term. In practice, we believe that *dotfile* owners *want* their files to be copied and adapted by others, so they will often store them in a *dotfiles* repository under a version control system (VCS) such as *Git*, and host the repository publicly on code hosting platforms such as *GitHub*.

In this work, we study *dotfiles* that have been publicly shared on *GitHub* to better understand the practice of maintaining user-configuration files. We collect the *dotfiles* repositories from *GitHub* using popularity and activity metrics with the help of *GHTorrent* [57]. Specifically, our study investigates three research questions:

RQ1: Who are the owners of *dotfiles* repositories?

Here, we study the prevalence of sharing *dotfiles* among developers to understand how widespread the practice is. We wish to confirm if it is software developers — rather than casual users — who are doing the sharing, since *dotfiles* can also be used for more general-purpose software used by the broader public. To answer this RQ, we first traced the occupation of the owners of *dotfiles* repositories collected by their public profile; we observed that the majority of them appear to work in a field that involves programming activities. We then checked the top 500 most-starred users on *GitHub*, and we found that 129 (i.e., 25.8%) of these top users own a variant of a *dotfiles* repository on *GitHub*.

RQ2: What kind of user-specific configuration files do users track in their *dotfiles* repositories?

In this RQ, we hope to build an understanding of which specific configuration files developers manage and track with *dotfiles* repositories. We extracted the *dotfiles* by their normalized names (e.g., adjusted for different folder structures) from the *dotfiles* repositories, and created a taxonomy of the top fifty most common *dotfiles* in our dataset. We find that configurations for text editors and **NIX* shells are most common in *dotfiles* repositories. Meanwhile, meta-files such as *README* files, software license information, and deployment scripts (e.g., `setup.sh`, `Makefile`) are also common.

RQ3: How do developers update their *dotfiles*?

In this RQ, we aim to understand how developers maintain their *dotfiles*. We sampled 400 commits uniformly at random in *dotfiles* repositories, and performed an open card-sort to infer the intent of the commit. We then selected the most frequently edited *dotfiles* in all *dotfiles* repositories and modeled their code churn history as time-series. We used the state-of-the-art time-series clustering technique [105] to extract patterns of code churn history of frequently edited *dotfiles*. We find that 54.8% of commits directly change the configuration of software tools, meanwhile, other commits focus on managing the *dotfiles* repository such as updating documentation, adjusting deployment scripts. We observe that

all types of *dotfiles* can be found in every code churn history pattern, suggesting developers play a more important role than the type of *dotfiles* in the frequency of editing *dotfiles*.

The key contributions of this work includes (1) collecting a set of quality shared *dotfiles* repositories (available in replication package); (2) providing empirical evidence in the prevalence of *dotfiles* sharing among developers; (3) providing a taxonomy of common *dotfiles*; (4) identifying the intent of commits in *dotfiles* repositories; (5) extracting code churn history pattern of frequently edited *dotfiles*.

5.2 Data collection

The corpus of *dotfiles* repositories used in this study is collected from *GitHub*. We used the *GHTorrent* [57] data from late 2019 to discover *dotfiles* repositories, and we cloned a filtered set of *dotfiles* repositories for analysis in mid-2020.

Managing *dotfiles* repositories is mostly a convention among developers. There exists no universal method to manage a *dotfiles* repository, thus, it is impossible to extract all *dotfiles* repositories from *GitHub*. Upon inspection on common *dotfiles* repository naming conventions, we decided to selected all repositories with the project name matching exactly the string "dotfiles" in the GHTorrent database. We found 167,663 *dotfiles* repositories from the query.

Since *GitHub* is known to be filled with low quality projects (e.g., temporary student projects that are unmaintained) [72], we performed further filtering with three criteria to ensure the quality of the collected *dotfiles* repositories:

1. ≥ 5 stars — to include *dotfiles* repository with endorsement by other users
2. ≥ 5 commits in the *dotfiles* repository— to filter out inactive *dotfiles* repositories
3. ≥ 10 commits in other repositories — to filter out inactive users on *GitHub*

Our strict filtering is intended to ensure that the *dotfiles* repositories we collect belong to software developers. Since some *dotfiles* do not necessarily correspond to development tools (e.g., configuration for window managers), we want to avoid *dotfiles* repositories that focus on these aspects (e.g., purely aesthetic customizations [116]). Therefore, we included the third criteria to exclude *dotfiles* repositories where the owner do not have other activities on *GitHub*. We further investigate the owners of *dotfiles* repositories in

RQ1 where we traced the profession of *dotfiles* repositories owners by their shared public profile.

After the filtering process, we have 3,757 *dotfiles* repositories. We then continued to clone these repositories from *GitHub*. Due to the time difference between the *GHTorrent* dataset (est. late 2019) and the time of cloning (est. mid 2020), a small proportion repositories are no longer available. Some common reasons includes migrating to another platform (e.g., *GitLab*) and *DCMA* take-down requests.

In the end, we are able to retrieve 3,305 *dotfiles* repositories from *GitHub*. In the rest of the paper, we refer to these *dotfiles* repositories as the *dotfiles dataset*.

5.3 Results

In this section, we answer the research questions we proposed, and we make several observations about the sharing of *dotfile* repositories.

5.3.1 RQ1: Who are the owners of *dotfiles* repositories?

Motivation Sharing *dotfiles* is often endorsed by the community [64] as a best-practice for software developers. However, the scale of *dotfile* sharing in practice remains unclear. Moreover, software development tools are not the only kind of applications that store configuration information in *dotfiles*. Many non-development focused tools — such as **NIX PDF* readers — also store their configuration in *dotfiles* [116]. While these applications are still important for developers, they do not focus of software development directly. Thus, in this RQ, we aim to understand the popularity of sharing *dotfiles* among developers, and evaluate our *dotfiles dataset* by verifying whether the *dotfiles* repositories are owned by developers.

Methodology To achieve a border understanding, we approach the owners of *dotfiles* repositories from two perspectives, the popularity of owning a *dotfiles* repository among the most-starred users on *GitHub*, and whether the selected *dotfiles* repository owners are developers.

To understand the popularity of sharing *dotfiles* among developers, we checked for the top 500 most-starred users on *GitHub* to see if they own a variant of a *dotfiles* repository.

In September 2022, we collected the information of the top 500 users by leveraging a community tracker [71]. With the user ID collected, we used the *GitHub* API to retrieve the list of repositories owned by each user in our list. For each user we checked whether a *dotfiles* repository exists by performing a fuzzy string match between the repository name and the term "dotfiles". The fuzzy algorithm works by calculating the Levenshtein distance [78] between the substrings. Based on manual inspection, we selected 85 as a cut off score based on fuzzy matching, where slight variations of "dotfiles" is accepted (e.g., "dotfiles-osx", "dotfiles-local"). To avoid situations where a repository's name contains only a few characters and overlaps with "dotfiles" (which would give a perfect score), we set an extra requirement for the repository name to be at least length of four.

We sampled 100 *dotfiles* repositories uniformly at random from the *dotfiles dataset* to try to infer the profession of the owner. For each repository, we start by visiting the owner's *GitHub* profile homepage. When the user provides additional information such as a personal profile, personal website links, and *LinkedIn* profile page, we followed the publicly available information and took note on the owner's profession, if we could find it. When the information was not present, we supplemented the process with a simple web query based on the their *GitHub* user name. In these cases, we can sometimes discover the owner's information as people tend to use the same user name across platforms. We restrained ourselves from further investigations (e.g., leveraging the author name and e-mail) to avoid violating the users' privacy.

Results We now report the findings we gathered from investigating the most-starred users on *GitHub*, and from tracking the public profiles of the owners of *dotfiles* repositories contained in the *dotfiles dataset*. We start by making the following observation:

Observation 1: Developers often share *dotfiles*. Owning a *dotfiles* repository is common among the most prolific developers on *GitHub*. In 129 (i.e., 25.8%) of the top 500 most-starred users have a *dotfiles* repository. We note that the actual number may be higher than the measured one, since the "user" here may represent a organization instead of a single developer.[1] In Fig. 5.2, the solid blue line represents the cumulative number of users with a *dotfiles* repository in the top 500 most-starred users. That is, at the 100 mark on the x-axis, it represents the number of users with a *dotfiles* repository in the top 100 most-starred users. As shown in the figure by the dotted red line, we observe a linear growth in the number, suggesting the prevalence of sharing *dotfiles*.

We found that the majority of the owners of *dotfiles* repositories in the *dotfiles dataset* appear to work in a programming-related job. As shown in Table 5.1, 79 *dotfiles* repositories

[1] https://docs.github.com/en/get-started/learning-about-github/types-of-github-accounts

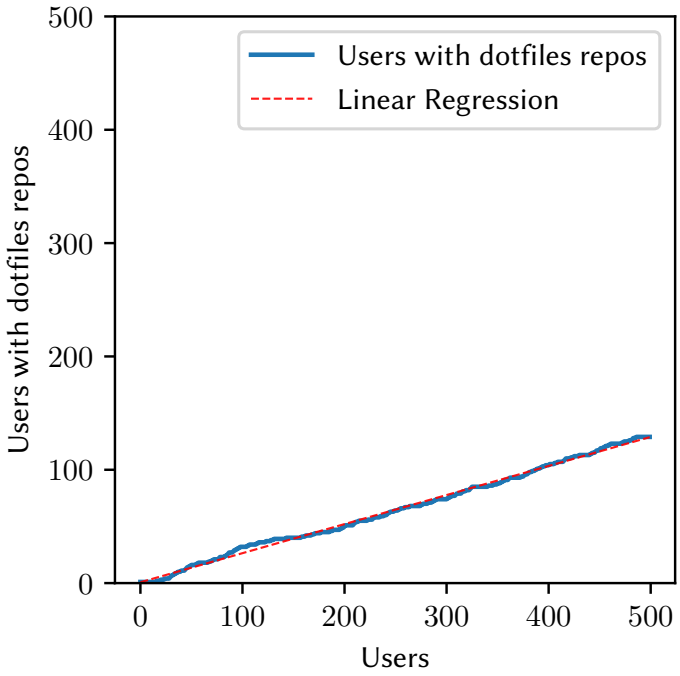


Figure 5.2: Number of top users by total repo stars on *GitHub* with *dotfiles* repositories

owners are software developers, with 3 system admins, 6 students, and 3 researchers. Unfortunately, we were unable to identify the profession for 9 of the *dotfiles* repositories owners. This finding gives us high confidence that the *dotfiles* contained in the *dotfiles dataset* are mostly from developers.

Table 5.1: Sampled 100 *dotfiles* repositories owners by occupation

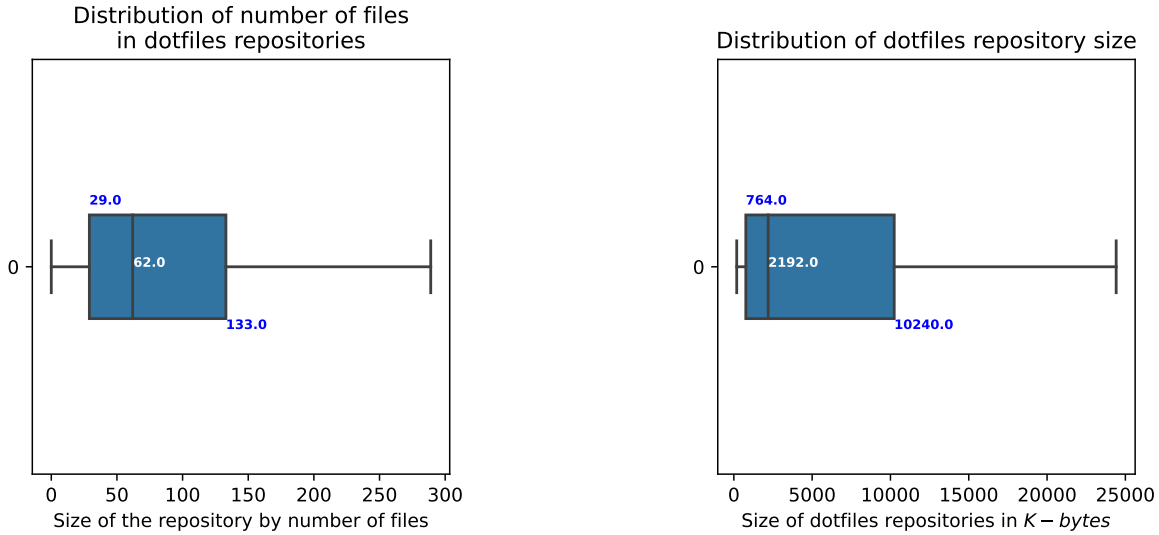
Self-declared profession	# of owners of *dotfiles* repositories
Software developer	79
System admin	3
Student	6
Researcher	3
Unknown	9

We believe the prevalence of sharing *dotfiles* comes from the familiarity that developers have during their regular job to use version control systems. Similar to source code, the plain text property of *dotfiles* makes tracking them the same as tracking source code. As a result, developers can easily manage their *dotfiles* in the same way they manage source code.

5.3.2 RQ2: What kind of user-specific configuration files do users track in their *dotfiles* repositories?

Motivation After confirming the *dotfiles dataset* from developers, in this RQ we explore the content contained in *dotfiles* repositories. Since *dotfiles* are user-specific configurations, we first aim to identify which software tools developers are explicitly maintaining configuration settings for. With an idea of what developers track in *dotfiles* repositories, we can understand the tools that are likely to be customized.

Methodology We extracted all unique *dotfiles* from the *dotfiles dataset*. We represent each *dotfile* by its lowercase base name, and remove the leading dot if any. For example, a file `vim/.VIMRC` (relative to *git* base) will be normalized to `vimrc`. We applied this step since the structure of each *dotfiles* repository is unique. Developers may have different approaches to managing *dotfiles* (e.g., writing deployment scripts, using *dotfiles* management



(a) Number of files in *dotfiles* repositories

(b) Size of *dotfiles* repositories

Figure 5.3: Content size of *dotfiles* repositories

tools), so the normalization step ensures that we are correctly tracking the *dotfiles* across different structures. We also ignore the file extension for *README* files to accommodate the developers' choice of format; thus, *README* files in *Markdown*, *Org*, *Asciidoc*, etc. are treated uniformly. After we have the normalized *dotfiles*, we create a taxonomy of *dotfiles* based on the top 50 most common *dotfiles*. The most common *dotfiles* are determined by the percentage of presence in *dotfiles dataset*.

Results Fig. 5.3 provides a overview of the size of *dotfiles* repositories. The median number of files in *dotfiles* repositories is 62, and the median repository size is 2Mb.

In Table 5.2, we list the top 20 with the percentage of *dotfiles* repositories they are present in the *dotfiles dataset*. In addition to project meta files (e.g., *README*, *license*), we found that configuration files for *Vim*, *Git*, *tmux*, *zsh*, and *bash* to be the most common.

Observation 2: Configurations for shell and text editors are the most common. Configurations for shell also contribute to the top 50 list. We observe configuration files associated with three different shells, namely *zsh*, *bash*, and *fish*. Auxiliary files for shells such as `aliases`, `profile`, and `zshenv` are also common. These files complement the shell configurations by splitting the configuration based on functionality (e.g., separate aliases) and environment (e.g., login and interactive sessions).

Table 5.2: 20 most tracked *dotfiles* by popularity

Filename	Associated application	Application type	#(%)
`README`		meta	2923 (88.4%)
`gitignore`	*git*	version control	2672 (80.8%)
`vimrc`	*vim*	text editor	2113 (63.9%)
`gitconfig`	*git*	version control	2087 (63.1%)
`tmux.conf`	*tmux*	terminal multiplexer	1950 (59.0%)
`zshrc`	*zsh*	shell	1856 (56.2%)
`config`	multi*		1486 (45.0%)
`bashrc`	*bash*	shell	1342 (40.6%)
`gitmodules`	*git*	version control	1131 (34.2%)
`bash_profile`	*bash*	shell	954 (28.9%)
`init.vim`	*vim*	text editor	904 (27.4%)
`license`		meta	844 (25.5%)
`inputrc`	*readline*	text edit	798 (24.1%)
`xresources`	*Xorg*	display server	709 (21.5%)
`install.sh`		meta	698 (21.1%)
`gemrc`	*Ruby*	package manager	625 (18.9%)
`xinitrc`	*xinit*	start *Xorg*	601 (18.2%)
`gitignore_global`	*git*	version control	537 (16.2%)
`zshenv`	*zsh*	shell	523 (15.8%)
`aliases`	shell	shell	510 (15.4%)

**:* `config` is a fairly common name selected by multiple software developers as their default configuration file name. Some common examples include *i3wm* a window manager, `/.config/i3/config`, and *fcitx5* a input method manager, `/.config/fcitx5/config`.

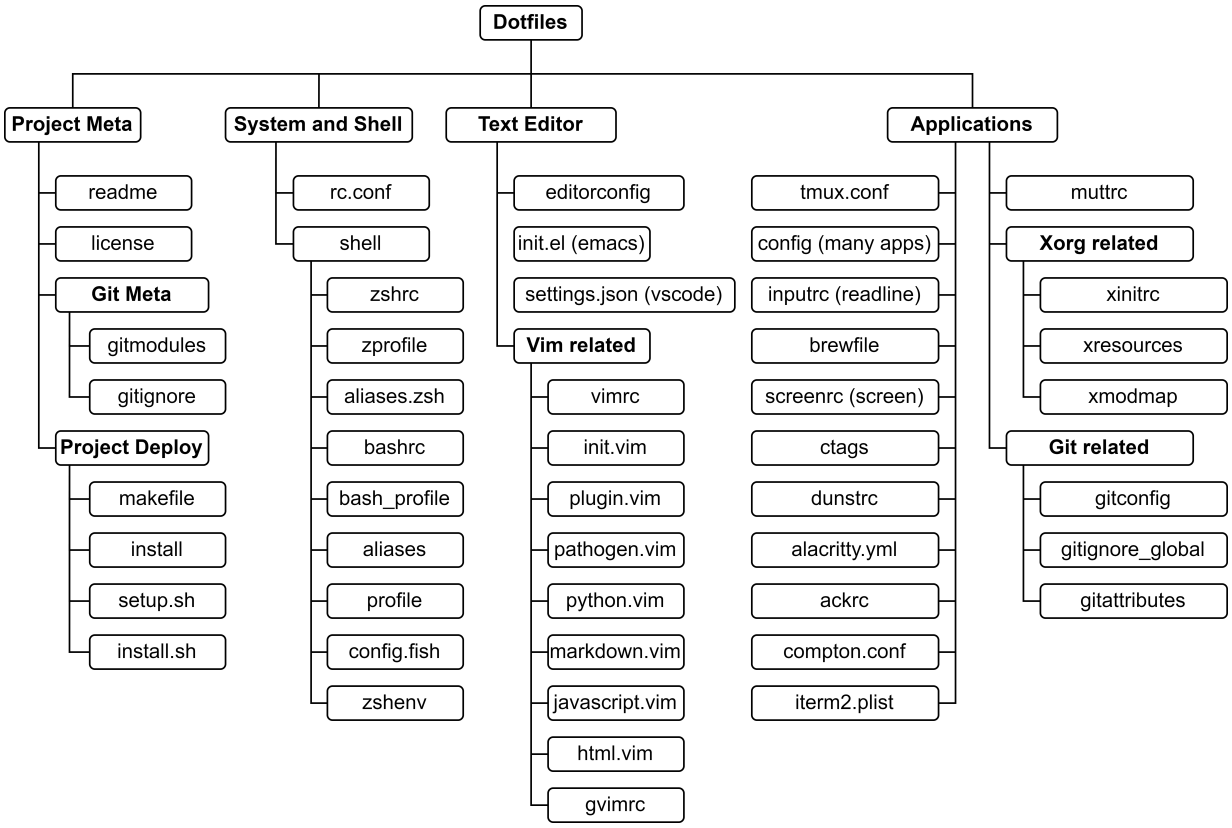


Figure 5.4: Taxonomy of the top 50 most common *dotfiles*

Another major contributing factor to the top 50 list is the collection of text editor related configurations. *Vim*, *Emacs*, *VSCode*, and a universal *editorconfig* can be observed from the list. *Vim* allows file type specific setups in separate files, which explains the 6 different `.vim` files associated with different languages. The remaining 3 files correspond to variants of *Vim*, the default (`.vimrc`), the graphic version of *Vim*— *GVim* (`gvimrc`), and a major fork of *Vim*— *NeoVim* (`init.vim`). Despite *VSCode* being the most popular text editor [103], the popularity of `settings.json` is overshadowed by *Vim* in our study. Our conjecture is that *VSCode* encourages users to use its builtin synchronization support, removing the need for managing the configuration through a *dotfiles* repository.

Observation 3: Meta-files are common in *dotfiles* repositories. We also want to highlight the *dotfiles* repository specific files in the top 50 list. Commonly found in *GitHub* projects, *README* and `license` are also common in *dotfiles* repositories. The *README* file often contains information about the repository and instructions to deploy the *dotfiles* in a new environment. The detailed deployment method can leverage an existing *dotfiles* management system, such as *Chezmoi*[2] or *GNU stow*, or with custom install scripts as `install.sh` or `Makefile` which both show up in the top 50 list.

Observation 4: GUI configurations are rare in *dotfiles* repositories. In Fig. 5.4, we show the taxonomy of the 50 most popular *dotfiles*. We can see immediately that most *dotfiles* are for command line applications, while only a few *dotfiles* are for *GUI* application. For example, `iterm2.plist` for *iTerm* on *MacOS*, and `dunstrc` for the notification manager *dunst*. There are also a couple of files related to the *Xorg* display server, such as `xinitrc` and `xinit`. The *GUI*-specific *Vim* configuration `gvimrc` is also present in the list. We suspect the lack of *GUI* configuration files is because — unlike in the 1980s **NIX* world — modern *GUI* applications are often configured using a *GUI* interface, and the settings are managed and stored differently.

This makes managing the configuration files hard. Some applications also have their own mechanism for managing configuration (e.g., web browsers). We note that despite *Windows* being common for development [103], we did not observe any *Windows*-specific software configuration files in the top 50 list. We observed some instances of *AutoHotKey* scripts which is a popular Windows application for automation. We suspect the lack of *Windows*-specific configuration files is due to *Windows* prefers using the *registry* database in which applications and system components store and retrieve configuration data [89].

[2] https://github.com/twpayne/chezmoi

5.3.3 RQ3: How do developers update their *dotfiles*?

Motivation We theorize that developers often update their *dotfiles* to adjust the configurations to adapt to both the change in development need (e.g., switching to a new language) and also the change in personal taste (e.g., testing out alternative tools). The *dotfiles dataset*, which contains real-world *dotfiles*, allows us to investigate the details of how developers update their *dotfiles*.

Through studying the change history of *dotfiles*, we can understanding of how developers maintain their *dotfiles* repositories. Also, through studying the frequently edited *dotfiles*, we can better grasp the patterns of how developers update the *dotfiles*. Our theory is that different types of *dotfiles* may be updated differently. For example, after a rapid period of changing the *README* file, the information will stabilize and require only infrequent updates subsequently. Meanwhile, for other tools such as *Emacs*, the developers may fine tune it frequently to adapt the tool to their needs.

In this RQ, we study the intent of *dotfiles* repository commits and the code churn history patterns to better understand how developers update their *dotfiles*.

Methodology We start by sampling 400 commits from the *dotfiles dataset*. We then performed open card-sorting on the commits to derive the intent of the commit. We went through the commits together and grouped similar commits. We iterated on the groupings until we felt that no commits should be moved to another group. This step allows us to build an understanding of why *dotfiles* are updated by developers.

With the next step, we attempt to discover historical patterns in code churn. We tracked the total number of commits for each *dotfile* in the *dotfiles* repository. We extracted frequently-edited *dotfiles* that have at least 20 commits and are at least 1 year old. We modeled each of the frequently updated *dotfiles* as a time-series using its code churn history. The time-series is represented by cumulative total code churn summed in each commit (i.e., sum of added and deleted lines). Since most time-series analysis techniques require regular time-series with the same number of timesteps, we limit the time-series from the last step to only the first year, and normalize the timesteps to 1 day. At this step, each time-series have 365 data points. We then removed time-series which have less than 20 updates. In other words, the resulting time-series are from *dotfiles* that have at least one year of history and have at least ten commits associated with it from ten different days in the first year. In the end, we have a total of 12,502 time-series representing 3090 distinct *dotfiles* in the frequently updated set of *dotfiles*.

We then leverage a time-series clustering method called *K-Shape* [105] to extract the patterns of code churn history. We first normalize the time-series with a mean-variance filter; this filter allows us to remove the bias from absolute change of the time-series and focus more on the relative change. For example, when two files share the percentage of change (e.g., 10%) while one file is 500 lines and the other is 50, the mean-variance filter is able to normalize the two time-series.

We then leveraged the *K-Shape* clustering algorithm for time-series clustering. The *K-Shape* algorithm uses metrics that focus on the shape of the time-series. This approach has been found to have similar performance to DTW [123], which, in turn, has been shown to outperform Euclidean distance, and is more computationally efficient than DTW [56].

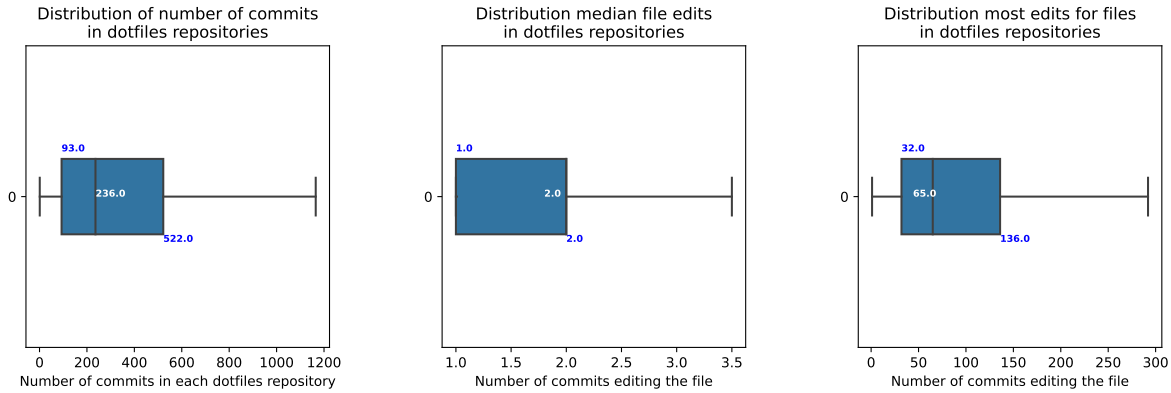
K-Shape, which is derived from *K-means* [84], also suffers from the problem of determining the proper k value. Since the distance between time-series, and determining the centroid of a series of time-series is still an on-going challenge in time-series analysis, there is no universal method to validate the result of the clusters [2]. So we experimented with different k values ranging from 2 to 10, and selected the best k value (i.e., 4) base on our interpretations of the clustering results.

Results In Fig. 5.5, we show the distribution of the number of commits, the median commits by file per *dotfiles* repository, and the number of commits for the most edited file per *dotfiles* repository. As suggested by Fig. 5.5a, the *dotfiles* repositories are updated frequently (note that in our filtering process, we only require the *dotfiles* repository to have five commits).

Observation 5: The majority of *dotfiles* receive at most one update. As shown in Fig. 5.5b, in most *dotfiles* repositories, the median number of commits per file is only two. Since the *dotfile* is introduced to the repository in the first commit, it means that the *dotfile* have only received one update since its introduction. This suggests that a large proportion of the *dotfiles* repository focuses on "cold storage" where the content rarely gets updated.

Observation 6: Most *dotfiles* updates focus on a small set of files. Although most *dotfiles* seldom receive updates, the most frequently updated *dotfile* in each *dotfiles* repository receive many updates as shown in Fig. 5.5c. While most *dotfiles* remain stable, developers update some *dotfiles* frequently.

Observation 7: Most *dotfiles* repository commits relate to tweaking the behavior of software tools. We found that the majority (i.e., 63.3%) of the *dotfiles* repository commits are related to tweaking the behavior of software tools. We also found that 30.8% of the commits edited configurations in the form of scripting (e.g., improving



(a) *dotfiles* repositories commits

(b) *dotfiles* repositories median file updates

(c) *dotfiles* repositories most updated file

Figure 5.5: Information on commits for *dotfiles* repositories

a shell script), 24.0% of the commits changed parametrized options (i.e., a predefined configuration option), and 8.5% of the commits are related to creating and modifying shortcuts (e.g., adding a *Git* alias). The details of the card-sorting results are shown in Table 5.3. Scripting is often related to defining custom actions in configuring software. It can be most commonly found in shell scripts for automation (e.g., swapping the location of two files) and performing complex functionalities in text editors (e.g., run code formatters when a file is saved).

Table 5.3: Type of *dotfiles* commits

Type	#(%)
tweaking script behavior	123 (30.8%)
tweaking parametrized options	96 (24.0%)
dotfiles deployment management	41 (10.3%)
fixing bugs in configuration	37 (9.3%)
tweaking shortcuts	34 (8.5%)
refactoring configuration	17 (4.3%)
managing external resources	23 (5.8%)
documentation update	20 (5.0%)
misc	18 (4.5%)

Observation 8: A noticeable amount of commits focus on *dotfiles* manage-

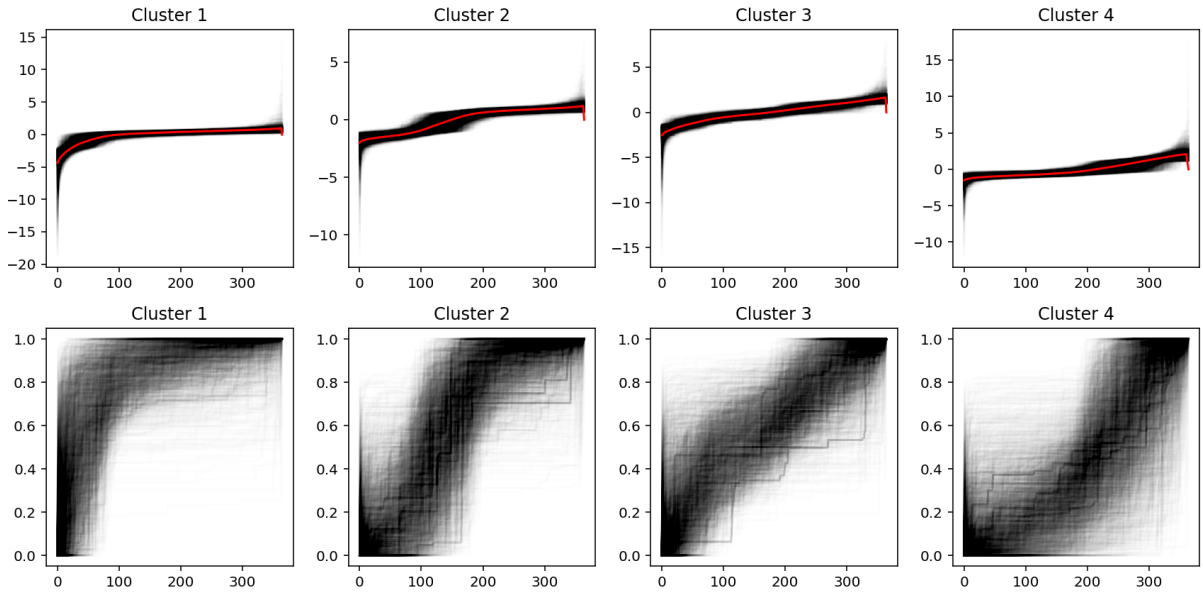


Figure 5.6: *K-Shape* clustering ($k = 4$) results on time-series modeled by code-churn history. Top: Mean-variance normalized. Bottom: Min-max normalized.

ment. We found that a noticeable amount of commits (i.e., 25.4%) focus on *dotfiles* management. Documentation updates occur in 5.0% of the commits where developers update the contents of *README* files. 10.3% of the commits directly involve changing the files related to *dotfiles* deployment. These commits modify deployment specific files such as `Makefile` and `setup.sh`. In 4.3% of the commits, developers refactor their *dotfiles*. This is often indicated by the commit message such as "clean up". We also observe 5.8% of the commits that deal with external resources. For example, some developers manage their *Vim* plugins with *Git* submodules leveraging package managers like pathogen.[3])

The rest of the commits (i.e., 4.5%) do not fit in either of the above mentioned categories. These commits are often aggregations of multiple changes and do not have a single purpose.

Observation 9: Three types of *dotfiles* code churn history patterns can be observed in the 4 clusters. In Fig. 5.6, we show the clustering results from *K-Shape* in both mean-variance and min-max normalized form. The figures on the top are the mean-variance normalized time-series. These time-series are the raw data fed to *K-Shape*. The red line is the average of the time-series in each cluster. The slope of the line after

[3] https://github.com/tpope/vim-pathogen

mean-variance normalization represents the change. So when the slope is near zero it means that there is limited code churn during the period. The figures on the bottom are min-max normalized between 0 and 1 for better visualize the code churn history. Cluster 1 are *dotfiles* that receive updates frequently after the *dotfile* is introduced to the *dotfiles* repository, but remain mostly unchanged since then. Clusters 2 and 4 are similar in the sense that these *dotfiles* receive updates over time, but differ when the majority of the updates occur. *dotfiles* in Cluster 3 are continually updated over time and the modified size are consistent throughout.

We also report here over observations on the clustering results in different k values. When we began with $k = 2$, it is immediately clear that one type of code churn history pattern is the *dotfiles* stops receive updates shortly after it has been introduced. As we gradually increase the value of k, we observe different patterns when do the update occur over time. Similar to Cluster 2, additional clusters in higher k value will have a sudden update in different time range. In other words, these clusters will overlap if we shift the time-series.

The distribution of the top 20 frequently updated files across the clusters is shown in Table 5.4. It is immediately clear that all types of *dotfiles* are well represented in each of the clusters. This suggests that the type of *dotfile* does not have a large effect on how the *dotfile* is maintained. We believe the developers have a larger impact on how the *dotfiles* are updated. We checked the frequently updated *dotfiles* by each developer and observe a median 57.1% of the developer's frequently updated *dotfiles* belong to one pattern. In other words, if the developer has no impact, this percentage will be around 25%. However, note that the median number of frequently edited *dotfiles* by developer is only three, so the data is likely biased. A developer who enjoys tinkering may constantly update their *dotfiles* to change their workflow. At the same time, a goal-oriented developer may spend some time to configure the tools at first to get it working and stick to the configuration afterwards.

5.4 Discussions and Implications

Through our study of *dotfiles* repositories, we have gained insights in how developers configure their software tools and how they manage the configurations in *dotfiles* repositories. Based on our observations, we discuss the challenges we perceive in managing *dotfiles* repositories. Moreover, we discuss the potential for how shared collections of real-world user-specific configurations can benefit developers, tool designers, and researchers.

Table 5.4: Distribution of frequently edited *dotfiles* across clusters

dotfile type	Cluster 1 #	%	Cluster 2 #	%	Cluster 3 #	%	Cluster 4 #	%	Total
`vimrc`	243	20.59%	325	27.54%	245	20.76%	367	31.10%	1180
`zshrc`	165	21.77%	194	25.59%	184	24.27%	215	28.36%	758
`readme`	63	17.50%	145	40.28%	69	19.17%	83	23.06%	360
`config`	95	26.61%	102	28.57%	75	21.01%	85	23.81%	357
`bashrc`	60	17.80%	103	30.56%	73	21.66%	101	29.97%	337
`gitconfig`	55	17.80%	73	23.62%	79	25.57%	102	33.01%	309
`tmux.conf`	59	19.60%	94	31.23%	65	21.59%	83	27.57%	301
`gitmodules`	42	14.63%	93	32.40%	70	24.39%	82	28.57%	287
`aliases`	37	17.21%	58	26.98%	41	19.07%	79	36.74%	215
`init.vim`	47	25.54%	49	26.63%	46	25.00%	42	22.83%	184
`install.sh`	37	26.81%	37	26.81%	27	19.57%	37	26.81%	138
`init.el`	34	25.00%	34	25.00%	20	14.71%	48	35.29%	136
`brewfile`	29	22.14%	36	27.48%	32	24.43%	34	25.95%	131
`bash_profile`	22	17.19%	18	14.06%	23	17.97%	65	50.78%	128
`gitignore`	32	25.20%	41	32.28%	15	11.81%	39	30.71%	127
`package.json`	8	7.14%	28	25.00%	43	38.39%	33	29.46%	112
`xinitrc`	15	15.96%	22	23.40%	29	30.85%	28	29.79%	94
`aliases.zsh`	25	27.47%	21	23.08%	24	26.37%	21	23.08%	91
`plugins.vim`	16	20.25%	19	24.05%	22	27.85%	22	27.85%	79
`makefile`	12	15.38%	23	29.49%	20	25.64%	23	29.49%	78

5.4.1 Challenges in Managing *dotfiles*

Challenge 1: Deployment of *dotfiles* requires effort. For the *dotfiles* to be read correctly by their associated software, the *dotfiles* need to be stored at the correct location. Common locations include the user's home directly and the `$XDG_CONFIG_HOME` (defaults to `$HOME/.config`). Users often have some kind automated setup for moving the *dotfiles* to the targeted location. The setup can be through a deployment script (e.g., `setup.sh`), or through dedicated tools such as *GNU Stow* [50]. In RQ2, we find many files among the most popular tracked *dotfiles* that focus on automating the process of deploying *dotfiles*. The deployment process can be either fully automated, where everything is taken care of by a script, or in documentation that contains the steps for deployment. Moreover, in RQ3, 10.3% of the commits deal managing the deployment of *dotfiles*. At the current stage, no universal method or tool exists for developers to manage their *dotfiles*.

The lack of standard method for managing *dotfiles* also introduces challenges in sharing *dotfiles*. For example, when a developer wish to explore other developers *dotfiles*, the developer must consult additional scripting and documentation to understand the setup of other developers. Future research can investigate this aspect further to better understand the challenges in deploying *dotfiles* and develop methodologies and tools to improve the process.

Challenge 2: *dotfiles* need to manage external resources. Similar to software development, we find that in *dotfiles*, developer also reply on external resources. Developers leverage *git submodules* to include external resources in their *dotfiles*. The external resources are plugins that extends the functionality of the software tools. The most common type of external resources we encountered in our study are *Vim* plugins. We believe its popularity largely comes from the popular *Vim* plugin managers *Vundle* and *pathogen*.

External resources can also be involved indirectly. Using *Vim* as another example, other plugin managers (e.g., *vim-plug*), instead of using *git submodules* requires only a declaration of the external resource by its URL to manage the extension. Unlike using *git submodules*, there exists a challenge in replicating the *dotfiles*. Base on the time of the deployment, different versions based on different commits will deployed.

Challenge 3: *dotfiles* can leak privacy information. Some configuration may need to deal with sensitive information. For example, setting up software with API keys. As pointed out by a recent study, security leakage can be a huge concern on *GitHub* due to accidentally committing confidential information [47]. We observed developers taking actions to mitigate this issue. For example, using local environment variable to avoid writing sensitive information in plain text.

5.4.2 Leveraging *dotfiles* as a Software Repository

Configurations can be as simple as a set of key-value pairs (e.g., shell aliases), or as complicated as writing code (e.g., shell scripts). With the complexity and diversity in software development, tools needs to provide customizability to support different user scenarios. From our qualitative results in RQ3, we find that the majority of the configuration updates are directly related to modifying the configurations.

One of the challenges of tool developers is to understand the user requirements. However, given the complexity and customizability of software tools, the requirements may be complex and not single purpose. One well known example is the concept of a bug becoming a feature. A tool may be used differently from how it was originally designed. We believe by leveraging the collection of *dotfiles*, we can begin to address the challenges.

Challenge 4: Configurations in *dotfiles* provides user usage data indirectly. The rich real-world use case information contained in *dotfiles* repositories provides valuable information to understand software usage. We have observed many successes in leveraging telemetry techniques to better understand user behavior and in return improve user experience in software products [185, 113]. However, telemetry is a highly disliked practice in open source software. Traditionally, we rely on active members who participate and contribute to discussions to move forward on user experience. And from time to time, we can observe patters of "scratching an itch" type of contribution made by other users. However, this means that problems that are not directly faced by the active members and/or the problems that are not severe enough to attract users with an itch will remain unaddressed. *Dotfiles*, while not a silver bullet to solving the need for telemetry, can act as a middle ground to offer more information in addition to the vocal majority. The additional information shared by the community can help with discussions to provide a broader but not absolute view of how the tools are used in the community. This information can guide the process of creating a "sane default" setting. However, even if this information is provided, we may still face a divided community on what constitutes a "sane default". Some previous studies have explored the idea of leveraging community configurations to help with creating better configurations for production software [185, 143]. We believe that are unique challenges faced in *dotfiles*. Unlike software deployed in production environments (e.g., databases, web servers), user-facing software often does not have an optimal configuration.

Challenge 5: Configurations in *dotfiles* can help with creating advanced configuration recipes. Configurations for software tools can get complicated. For example, it is common to observe *Emacs* configurations with thousands of lines of *elisp* code. Developers often learn from other developers and gradually add and improve their own configuration. We believe that by leveraging the corpus of *dotfiles*, we can help create

better documentation to help developers configure their tools.

A similar concept can be observed in many successful open source libraries. A section called the "cookbook" can be find in the documentation. The section provides recipes on common use-cases, and serve as a base ground for developers to begin using the library. However, creating and maintaining the recipes can be a challenging tasks. We believe that *dotfiles*, can be a valuable source for creating "cookbook" recipes for software tools. By extracting common real-world configurations from *dotfiles*, we can improve the examples in documentation. For example, by extracting common configurations for editing Python from *Vim* configurations, we can provide extended real-world examples to help future developers to configure the software.

5.5 Threats to Validity

» *Internal Validity* — When deciding to create the *dotfiles dataset*, we set strict filters to the repositories selected. This suggests that we can miss out on repositories that may be of interest. However, we selected these criteria to ensure that we are looking at the *dotfiles* repositories owned by developers, which is confirmed by our results for RQ1. During our process to determine the occupation of *dotfiles* repositories owners, we leveraged only the owner's public profile as well as a simple user name search; in only a few cases were we unable to determine the occupation to our satisfaction. It is possible that the owner may as well work in fields that require developing software. Since a more thorough search, such as using the owner's commit email, may violate the privacy of the owner, we decided stopping at the stage of viewing the public profiles.

We chose to leverage the *K-Shape* algorithm to extract the *dotfiles* maintenance patterns modeled as a time series based on historical code churn. *K-Shape* focuses on the shape attributes of the time-series and is an alternative approach compared to extracting features from the code commits. Using a shape-based algorithm allows us to avoid the downsides and potential problems of determining the important features to extract, and also have a higher focus on the activity trends of the code change alone. Since this is still a fairly new approach, in future work, research can investigate the differences and effectiveness of the different approaches for clustering code churn.

» *External Validity* — We have chosen *GitHub* as the source to collect the *dotfiles* repositories. While *GitHub* is largely considered as the *de facto* location for developers to host their projects, in recent years, many other options have gained popularity. Some options

are also centralized hosting a large variety of projects such as *GitLab*. Meanwhile, self-hosted options also exist, where an independent *GitHub*-like instance can be hosted for a specific developer or organization. One example would be *Savannah*[4], the place where *GNU* software is hosted. Due to security reasons, developers are also unlikely to host their work-related user-specific configuration files on publicly. Therefore, our results likely generalize only to the use of user-chosen software and have a bias towards open source applications.

Because they originate in the **NIX* world of the 1970s — largely before the advent of graphical user interface — *dotfiles* have historically had a heavy focus on the *CLI* applications. This can also be observed from the set of common *dotfiles* across repositories. Since *GUI* applications may have different ways of storing configuration (e.g., a binary file), and have built-in syncing functionality (e.g., *VSCode*), they may not be captured in the *dotfiles* repositories. Storing configurations in plain-text may be replaced by other means in the future, however, we believe the requirement for customization still exists. Future research and build specific tools to analyze the user-specific configuration files for *GUIs* and compare the results with our work.

5.6 Summary

In this chapter, we study the practice of sharing and maintaining *dotfiles* based on *dotfiles* repositories collected from *GitHub*. We observe that sharing *dotfiles* is a common practice among developers. Configuration files for text editors, shells and *Git* are the most common. While developers track many *dotfiles*, only a small amount of the *dotfiles* are constantly updated. We extracted code churn history patterns from frequently updated *dotfiles* and find that there is no significant relationship between the code churn history pattern and the type of *dotfiles*. We discuss the challenges developers face in managing *dotfiles* and how we can leverage the publicly shared *dotfiles* to help creating "sane defaults" and constructing "cookbook" recipes for documentation.

[4] https://savannah.gnu.org/

Chapter 6

Understanding App Stores the Software Engineering Perspective

App stores serve as the location where end users browse, discover, and manage their applications for a particular platform. The structure of app store where an organization oversees the operation, has been a novel way for software distribution with mobile platforms gaining popularity. App stores have become a rich software repository containing software artifacts from both the developer and user end. For example, studies have shown that developers use app store as a source to collect information about competitive apps and improve their own app through app product homepages and user reviews. While, we have seen app stores beyond the mobile platform (e.g., *Steam*), research have mostly focused on mobile app stores. In this chapter, we present an empirical study to understand the concept of app stores in the perspective of software engineering.

Related publication An earlier version of the work is currently under major revision for Empirical Software Engineering.

> Wenhan Zhu, Sebastian Proksch, Daniel M. German, Michael W. Godfrey, Li Li, Shane McIntosh. 2023. What is an App Store? The Software Engineering Perspective

6.1 Introduction

The widespread proliferation of smartphones and other mobile devices in recent years has in turn produced an immense demand for applications that run on these platforms. In

response, online "app stores" such as Google Play and Apple's App Store have emerged to facilitate the discovery, purchasing, installation, and management of apps by users on their mobile devices. The success of mobile app stores has enabled a new and more direct relationship between app creators and users. The app store serves as a conduit between software creators (often, developers) and their users, with some mediation provided by the app store. The app store provides a "one-stop shopping" experience for users, who can compare competing products and read reviews of other users. The app store also acts as a quality gatekeeper for the platform, providing varying levels of guarantees about the apps, such as easy installation and removal, expected functionality, and malware protection. To the software creator, the app store provides a centralized marketplace for their app, where potential users can find, purchase, and acquire the app easily; the app store also relieves the developer from basic support problems related to distribution and installation, since apps must be shown to install easily during the required approval process. Indeed, one of the key side effects of mobile app stores is that it has forced software developers to streamline their release management practices and ensure hassle-free deployment at the user's end.

The success of mobile app stores has also led to the establishment of a plethora of other kinds of app store, often for non-mobile platforms, serving diverse kinds of user communities, offering different kinds of services, and using a variety of monetization strategies. Many technical platforms now operate in a *store-centric* way: essential services and functionality are provided by the platform while access to extensions/add-ons is offered only through interaction with the app store. When new technical platforms are introduced, an app store is often expected to serve as a means to host and deliver products to its users [39]. Example technical platforms that use app store-like approaches include Steam [148], GitHub Marketplace [53], the Chrome Web Store [54], WordPress [164], AutoDesk [13], DockerHub [40], Amazon Web Services (AWS) [8], Homebrew [117], or Ubuntu Packages [23].

For platforms that operate in this way, the app store is an essential part of the platform's design. For example, consider source code editors, such as VSCode Marketplace and IntelliJ. The tool itself — which we consider to be a technical platform in this context — offers the essential functionality of a modern source code editor; however, many additional services are available through the app store that are not included by default. Thus, extensions that allow for language-specific syntax highlighting or version control integration must be added manually by the user through interaction with the tool's app store. We conjecture that the app store has fundamentally changed how some classes of software systems are designed, from the overall ecosystem architecture of the technical platform to the way in which add-ons are engineered to fit within its instances.

In this work, we will explore the general space of app stores, and also consider how app store-centric design can affect software development practices. Previous research involving app stores has focused mainly on mobile app stores, often concentrating on properties of the apps rather than properties of the stores. For example, Harman et al. performed one of the first major studies of app stores in 2012, focusing on the BlackBerry App World [61]. However, concentrating the investigative scope so narrowly may lead to claims that do not generalize well across the space of all app stores. For example, Lin et al. found that reviews of games that appeared in mobile app stores differed significantly from the reviews of the same game that appeared within the Steam platform's own app store [80]. In our work, we aim to take a more holistic approach to studying app stores by considering both mobile and non-mobile variants. In so doing, we hope to create a more general model of app stores that fits this broader space.

To achieve a holistic view, we start from the definition of an app store. A precise definition of the term "app store" has been omitted in much of the previous research in this area. Currently, Google Play and Apple's App Store dominate the market and are the main targets of research on app stores; in the past, the BlackBerry App World and Microsoft's Windows Phone Store were also important players, but these stores are now defunct[1]. Wikipedia recognizes Electronic AppWrapper [162] as the first true platform-specific electronic marketplace for software applications, but the term became popular when Apple introduced its App Store along with the iPhone 3G in 2008. Since then, the term has largely come to refer to any centralized store for mobile applications. We present our own working definition of the term "app store" in Section 6.2.

The goal of this work is to survey and characterize the broader dimensionality of app stores, and also to explore how and why they may feed back into software development practices, such as release management. As a step toward this goal, we focus on two research questions (RQs) that aim to explore the space of app stores:

RQ1: *What fundamental features describe the space of app stores?*

To understand app stores, we first need a way to describe them. It would be especially useful if this description framework would highlight the similarities and differences of app stores. We start by collecting a set of app store examples, and then extract from them a set of features that illustrate important differences between them. We then expand this list of app stores with search queries to derive a larger set of example stores. We explicitly seek generalized web queries to broaden our search space beyond the common two major

[1]The Windows Phone Store was absorbed into the broader Windows Store in 2015.

mobile app stores of *Apple* and *Google*. By combining the web queries and the initial set of app stores, we selected a representative set of app stores and extracted their features. At the end, we surveyed app stores and derive a feature-based model to describe them; expanded the set of app store through web queries; and extracted features based on the model for representative set of app stores.

RQ2: *Are there groups of stores that share similar features?*

Despite the ability to describe individual stores, it is also important to understand the relationships between different stores. Having a understanding of the natural groupings can help us gain insights into the generalizability of results gathered for app stores. We perform a *K-means* [84] clustering based on the extracted features of the expanded set of app stores collected previously. The optimal k value is determined by the Silhouette method [120]. The clustering results suggest that there are 8 groups in the expanded set of app stores. The differences can be observed in the type of application offered, standalone or extension, and/or type of operation, business or community-oriented.

Our investigations of the groupings formed can be used to further evaluate the generalizability of app store research from one group to another; furthermore, our work can form the basis for subsequent study of app stores in general.

6.2 Working Definition of an App Store

Previous researchers have often taken a casual approach to defining the term "app store", when a definition has been provided at all. For example, in their survey paper, Martin et al. define an app store as "A collection of apps that provides, for each app, at least one non-technical attribute", with an app defined as "An item of software that anyone with a suitable platform can install without the need for technical expertise" [85]. However, we feel that this definition is too generous. For example, consider a static website called Pat's Apps that lists of a few of someone's (Pat's) favourite applications together with their personalized ratings and reviews; superficially, this would satisfy Martin et al.'s requirements as it is a collection of apps together with Pat's own reviews (which are non-technical attributes). We feel that this kind of "store" is outside our scope of study for several reasons: Pat's software collection is not comprehensive, it is unlikely that Pat provides any technical guarantees about quality of the apps, and a passive list of apps on a web page does not constitute an automated "store".

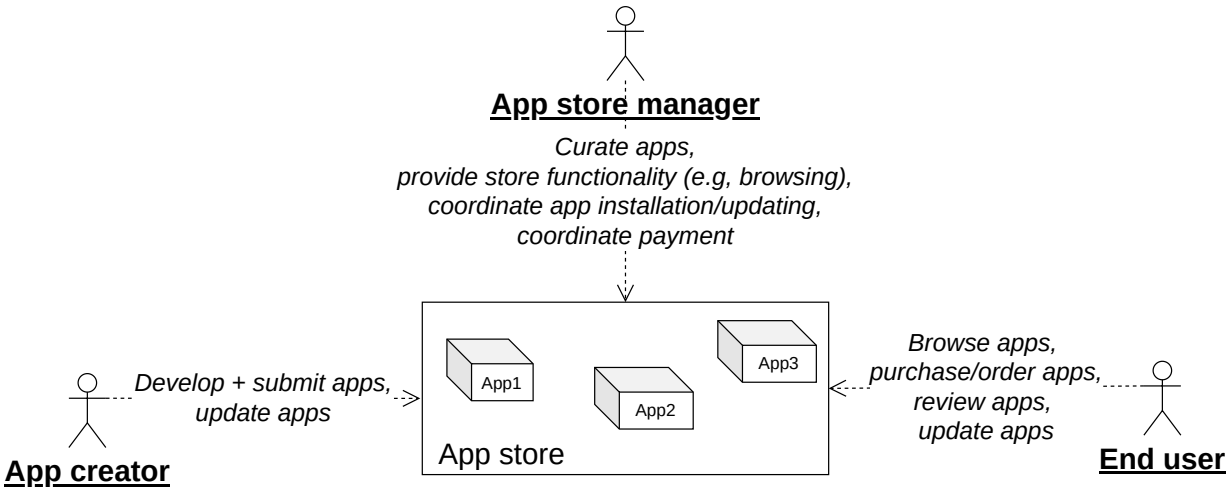


Figure 6.1: Three major stakeholders of most app stores

In our work, we seek to broaden the idea of app store beyond the well-known mobile ones. Because we are focused on exploring the notion of what app stores are, we formulate a working definition of the term; we did so to provide clear inclusion/exclusion criteria for the candidate app stores that we discover in Section 6.3.

Our working definition was influenced by considering the three major stakeholders of the app store model: the *app creators* who create and submit applications to the store; the *app stores* themselves, and the organizations behind their operation who curate the app collection and coordinate both the store and installation mechanisms; and the *end users* who browse, download, review, and update their applications through the app store (see Fig. 6.1).

We thus arrived at the following **working definition for *app store*** as an online distribution mechanism that:

1. offers access to a comprehensive collection of software or software-based services (henceforth, "apps") that augment an existing technical infrastructure (i.e., the run-time environment),
2. is curated, i.e., provides some level of guarantees about the apps, such as ensuring basic functionality and freedom from malware, and
3. provides an end-to-end automated "store" experience for end users, where
   (a) the user can acquire the app directly through the store,

(b) users trigger store events, such as browsing, ordering, selecting options, arranging payment, etc., and
(c) the installation process is coordinated automatically between the store and the user's instance of the technical platform.

We can see that using this working definition, our Pat's Apps example fails to meet all three of our main criteria.

We note that our working definition evolved during our investigations, and the above represents our final consensus on what is or is not an app store for the purposes of doing the subsequent exploratory study. The steps by which the representation is finalized is discussed in Section 6.3.1. For example, our working definition implicitly includes *package managers* such as the Debian-Linux apt tool and Javascript's NPM tool. It is true that package managers are typically non-commercial, and so are "stores" only in a loose sense of the term; furthermore, they usually lack a mechanism for easy user browsing of apps and do not provide a facility for user reviews. However, at the same time, they are a good fit conceptually: they tend to be comprehensive, curated, and offer an automated user experience for selection and installation. Furthermore, some package managers serve as the backend to a more traditional store-like experience; for example, the Ubuntu Software Center builds on a tool *aptitude*, which interacts with software repositories, to provide a user experience similar to that of Google Play.

6.3 Research Methodology

To investigate these research questions, we designed a three-stage methodology that is illustrated in Fig. 6.2. The goal of the first two stages is to answer RQ1, while the third stage addresses RQ2.

In the first stage (Step ① and ②) we identified our initial list of features using a small set of well-known app stores (Apple's App Store, Google Play, Steam etc.) During this stage, six individuals were involved in conducting the qualitative studies. In the second stage (Steps ③, ④, and ⑤) we methodically expanded our store list to a broad selection of 53 app stores that should represent a wide range of stores in practice. In this stage, two individuals were involved, unless otherwise specified. We then described these stores using the features identified in the first stage. A major goal of this stage was to evaluate whether the set of available features was sufficient to describe the characteristics of all these stores. This set of features forms the answer to RQ1.

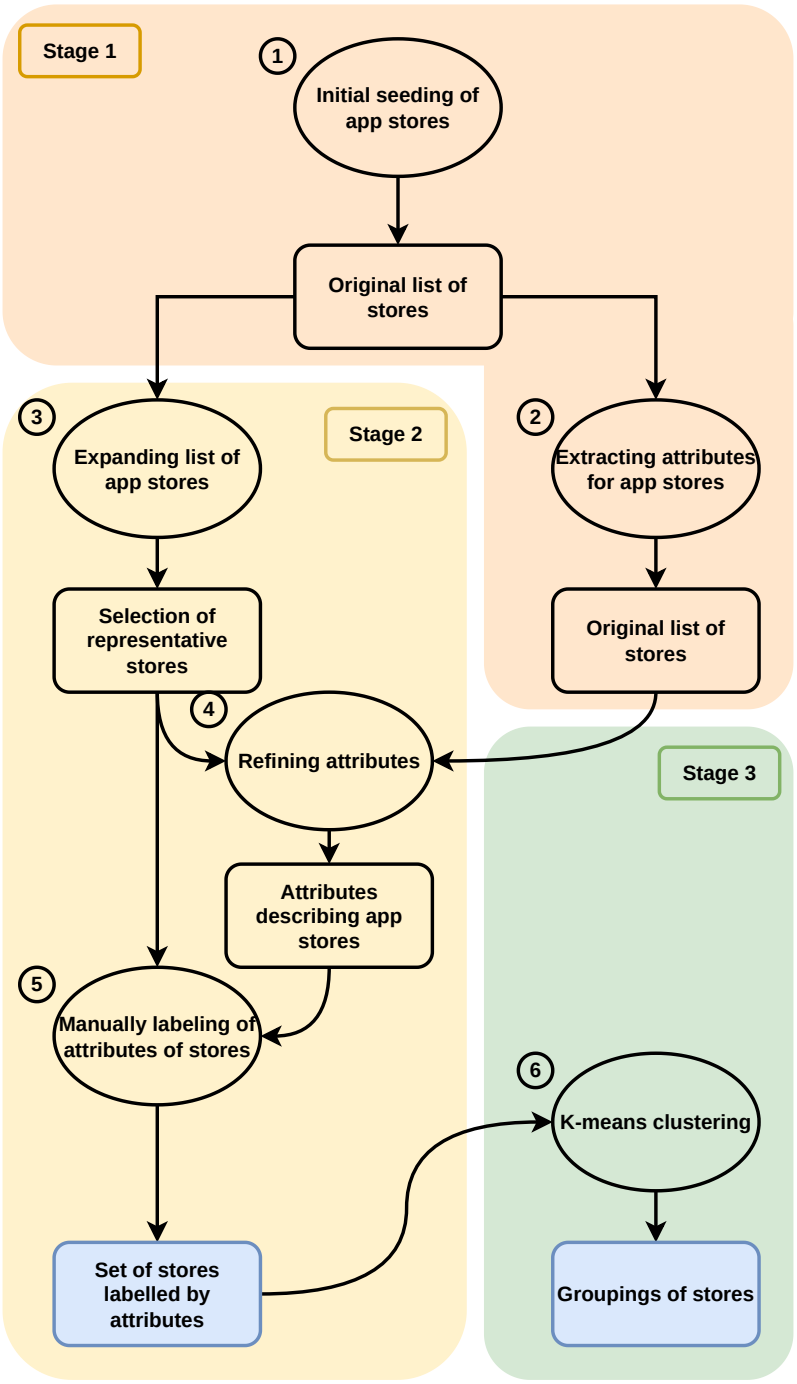


Figure 6.2: Methodology overview: There are three main *stages*, further broken down into six *steps*.

In the third stage (Step ⑥), we took advantage of the labeling of the 53 stores. We used *K-means* clustering analysis to identify groups of stores that shared similar features. These groupings form the answer to RQ2.

We now describe our methodology in more detail.

6.3.1 RQ1: *What fundamental features describe the space of app stores?*

Our basic assumption is that an app store can be categorized based on a finite set of features. The features would correspond to traits of the app store where they describe the distinguishing qualities or functional characteristics of the app store. We encode these features as binary values, i.e., each store *has* or *does not have* a feature.

In order to identify such features, we first created a seeding set of representative app stores. We started by enumerating well-known app stores that we were aware of (Step ①). Once this set of representative app stores was created, we used an iterative process to identify the features that we felt best characterized these stores (Step ②). We then used these features to describe each store.

Stage 1: Identifying features

First, we start with identifying representative characteristics of five stores and the possible features for each. We worked alone in this step; however, to seek better reliability as well as encourage diverse opinions, each store was assigned to two person. In total 15 stores were assigned where each store is assigned twice. We list the 15 stores with a short description in Table 6.1. After that, we met as a group to discuss their findings and further refine the proposed feature set.

In the subsequent iterations, we worked in pairs, and the pairings were reassigned after each iteration (Step ②). In these iterations, each pair was assigned a set of 2–3 app stores and was asked to describe them using the current set of features; a key concern was to evaluate whether the existing features were sufficient or needed refinement. For each store, each pair analyzed both its store-front and its documentation; in some cases, we could navigate the store as users but not as developers, in these cases, we relied on the store's supporting documentation.

After this step, we discuss the findings of each pair as a group and updated the set of features. The features were discussed in detail to ensure that they were conceptually

independent from each other. We also made sure that each feature applied to at least one store to ensure that it was relevant.

Our process leveraged ideas from the coding process of *Grounded theory* [152] to extract the features of app stores; and followed the practice of open card sorting [34] to create the categorized feature set. Similar to prior work[1, 63, 86], we followed practices of *Grounded theory*'s coding process to extract the features (where we consider codes as a specific feature of app store operation) and stopped when we reached saturation with no new features added after a new round of describing app stores. Similar to prior work[151, 26, 155], we applied card sorting to the collected features so inter-related features are grouped together. We formed a group in this process and discussed how different features belong to the same conceptual group and stopped when consensus was reached.

Table 6.1: Investigated stores for feature extraction

Store	Description
Google Play Store	The flagship app store for Android
Apple App Store	The app store for iOS devices
Samsung GalaxyApps	App store specifically for Samsung devices
GitHub Marketplace	Providing applications and services to integrate with GitHub platform
Atlassian Marketplace	Providing applications and services to integrate with various Atlassian products
Homebrew	Package manager for MacOS
MacPorts	A package manager for MacOS
Ubuntu Packages	Software repository for the Ubuntu Linux distribution, with a official front end Ubuntu Software Center
Steam	Gaming focused app store running on multiple operating systems (e.g., Windows, Linux)
Nintendo EShop	Provides applications for Nintendo devices (e.g., Nintendo Switch, Nintendo 3DS)
GoG	Gaming focused store focusing on providing DRM free games
JetBrains Plugin Store	Provides plugins to enhance the behavior of JetBrains IDEs
VSCode Marketplace	Provides plugins to enhance the editor
Chrome Web Store	Provides extensions to enhance Chromium based web browsers
AWS Marketplace	Provides servers and cloud services

Stage 2: Expanding our set of app stores and further evaluation and refinement the features

Once we had agreed on the features, our next goal was to verify that these features were capable of describing other app stores that were not part of the initial seed, or if features

were missing or needed refinement. We used a common search engine, *Google*, to expand our set of app stores in a methodical manner (Step ③). To achieve the goal of including a broad range of yet undiscovered app stores, we first derived general search terms by combining synonyms for "app" and "store". More specifically, we have built all possible combinations of the following terms to construct our search queries:

First half `software` , `(extension -hair -lash)` , `(addon OR add-on)` , `solution` , `plugin OR plug-in` , `install` , `app` , `package`

Second half `repository` , `shop`, `("app store" OR store)`, `("market place" OR marketplace)` , `manager`

For example, a concrete query could be created by combining `app` and `("app store" OR store)`. For some queries, it was necessary to refine the term to avoid noise in the results; for example, searching for the term `extension` would mainly return results related to hair product or eye lashes. In total, with 8 synonyms for `app` and 5 synonyms for `store` we were able to create 40 unique *Google* search queries. We felt confident that these search terms were representative when we found that the initial seed list had been exhaustively covered.

Our *Google* search was performed in November 2020. We queried and stored the search results for each search query. We classified each result as to whether or not it corresponded to an app store. We devised two inclusion criteria for this decision: 1) the store in question should offer software or software-based services, and 2) the store in question should offer an end-to-end experience for users (ordering, delivery, installation). We considered only direct hits to the store (e.g., product page), and we explicitly excluded results that contain only indirect references to a store, such as blog posts, videos, or news. Any disagreements were resolved through discussion. However, despite our initial effort of maintaining a clear set of inclusion criteria for app stores, several corner cases arose during the labeling process. We discussed these cases as they arose, and continually updated the inclusion criteria throughout the labeling process. Since this step was conducted by two people, in a few special cases no agreement could be reached, another person acted as a moderator and resolved the disagreement by a majority vote. Over time, the inclusion criteria and features evolved and eventually reached a stable state (by Step ④). Our final state of the inclusion/exclusion criteria is presented as the working definition for app stores defined in Section 6.2.

The classification of search results was stopped when a new results page did not contain any new links to app stores, or once all 10 retrieved pages were analyzed. Initially, 586

URLs were examined until a saturation of agreement was reached (90.7% agreement rate). I continued to label the rest. In the end, a total of 1,600 URLs were labeled. Multiple search results can refer to the same store; these duplicates were detected and eliminated by using the root domain of the URL. The most common duplicate references were found for the domains *google.com* (61), *apple.com* (22), and *microsoft.com* (18). In the end, we found 291 stores. We note that the exact number of unique stores may differ since two root domains can point to the same store, *kodi.tv* and *kodi.wiki*, or the same root domain may contain multiple stores, *chrome.google.com* and *play.google.com*.

In the next step (Step ⑤), we constructed and labeled a set of app stores based on our identified features from Step ②. We began from the URLs labeled in the last step and selected the first three occurring stores for each search term; this resulted in 104 URLs pointing to 48 unique stores. Two of the stores were inaccessible by the us (ASRock App Shop requires physical hardware and PLCnext Store's website was not responsive at the time of labeling) and removed from the list. In addition, we discussed several more stores that we felt deserved explicit investigation: AWS, Flatpak, GoG, MacPorts, Nintendo eShop, Steam, and Samsung's Galaxy Store. These are the stores that the we investigated in Step ② but did not show up in the first three occurring results from the search terms. Meanwhile, the added stores all show up in the list of 291 stores identified by all labeled URLs.

We thus selected and labeled a total of 53 app stores. This sample is non-exhaustive, but we believe that our wide range of search queries has created a representative sample of the population of app stores that enables our experiments.

We proceeded to describe 12 app stores, selected as the first from each search query, using the set of features. This was done to make sure there was consistency in the interpretation and use of each feature. After that, I labeled the remaining stores.

The outcomes of RQ1 were a list of features that describe the main characteristics of app stores grouped by dimensions, and a set of 53 App Stores, each labeled using these features.

6.3.2 RQ2: *Are there groups of stores that share similar features?*

With the outcomes of RQ1, we next performed a *K-means* clustering analysis to identify groups of similar stores.

Stage 3: cluster analysis

To identify related app stores, we decided to cluster them using the *K-means* algorithm (Step ⑥).

To prepare our labels for the *K-means* clustering process, we converted each label of the feature to a binary value: 1 if the store has the feature, and 0 if it does not. Having binary-encoded data ensured that we do not suffer from having categorical values that do not make sense in the scope of *K-means*. However, performing *K-means* on binary data can also be problematic, since the initial centroids selected will be binary. To mitigate this issue, we applied Principal Component Analysis (PCA) [163] to both reduce the dimensional space and to produce a mapping in the continuous range. We kept all principal components that explained a variance of at least 0.05. Finally, we used the Silhouette method [120] to determine the best number of clusters within a range of 1 to 20.

As an unsupervised method, the result of *K-means* provides only the clustering result with the stores in each cluster. We then further discussed the results of the *K-means* process and categorized the clusters by the properties of the contained stores. Following our discussion and categorization, we assigned groupings and names to each of the clusters.

6.4 Results

In this section, we present the results of each of the research questions. The results are organized based on the three stages discussed in Section 6.3.

RQ1: *What fundamental features describe the space of app stores?*

Stage 1: Features Characterizing App Stores

As discussed in Section 6.3.1, we derive a set of features and organizational categories that describe the set of studied app stores; the results of these efforts are summarized in Table 6.2. We have modelled the features as a binary representation; thus, each store either has or does not have this feature. We note that for some categories, the features are mutually exclusive; for example, in the category *Rights Management*, a store can have either *Creator managed DRM* or *Store-enforced DRM*, but not both. In other categories, an app store may have several of the features within a given category; for example, there may be several kinds of communication channels between users, app creators, and the store owner for a given app store. We now describe each high-level category in detail.

Table 6.2: Features for describing app stores

Feature	Description
Monetization	The type of payment options directly offered by the app store.
Free	Free as in in the product can be directly acquired
One-time payment	A single payment needed for the product
Seat-based subscription	The subscription is based on the number of products provided
Time-based subscription	A payment is needed by a set time interval (e.g.,, monthy, yearly)
Resourced-based subscription	A payment is needed by the amount of resource used (e.g., API calls, CPU time)
Micro-transaction	Additional payment can be collected based on additional feature offered in a product
Custom pricing (i.e., "Contact us for price")	The actual price is based on a per case situation; this happens mostly in business-focused app stores
Rights Management*	How does the store take care of DRM on the product provided.
Creator-managed DRM	No DRM is offered by the store and is taken care of by the creator
Store-enforced DRM	Store wide DRM for every product offered in the store
Do I need an account?*	Whether it is possible to use the app store without registration.
Account required	An account is required to use the store
No registration possible	The store does not have an account system
Some features requires registration	Some content of the store is locked behind an account, but the store can be used without one.
Product type	The type of product the store offers.
Standalone apps	The product operates by itself
Extension/add-ons to apps/hardware	The product acts as a feature extension to another application/hardware
Service/resources	The software product is a service
Package/library	The product is not an end-user product, but offers functionality to other products
Target audience*	The intended users of the app store.
General purpose	The app store is intended to be used by everyone.
Domain-specific	The app store have a specific focus and is very unlikely to be used by a normal person
Type of product creators	The type of creators who submits products to the app store.
Business	The creators mostly have a commercial or business focus
Community	The creators are from the community (e.g., open source developers)
Intent of app store	The reason why the app store exists from the app stores' perspective.
Community building/support	The app store aims to serve a technical community
Profit	The app store aims to earn money
Centralization of product delivery	The app store aims to provide a way for customer to gather apps in a centralized way
Expanding a platform popularity/usefulness	The app store aims to extend functionality from the platform it is based on
Role of intermediary	The role app store play between the creator of products and the customer of the app store.
Embedded advertisement API	Provides an advertisement method for creators to take advantage of
CI/CD	Offers continuous integration/continuous deployment for creators
Checks at run time	Provide checks when apps installed from the app store is ran
Checks before making available to the customer	Provide checks when an app is submitted to the app store for quality reasons
Composability*	The relationship between products provided in the app store.
Independent	The products in the app store are unrelated to each other
Vendor internal add-on/extension/unlock	Some products can be based on other products from the same creator (e.g., game DLC, app feature packs)
Package manager type of app relationship	A dependency relationship exists between products in the app store
Analytics	The type of analytical data provided by the app store.
Sentiment and popularity ratings	Information related to the popularity of a product (e.g., downloads, score ratings)
Marketing feedback	Information related to marketing for the creator (e.g., sales, conversion, retention)
Product usage data	Information related to the usage of the product. (e.g., logging, user profiling)
Communication channels	The methods where different parties of the app store can communicate with each other.
Documentation	Information related to the operation of the store (e.g., instructions to install applications)
Product homepage	A homepage for a specific product in the app store
Ratings	Any form of rating customers can give to a product (e.g., star, score, up/down vote)
Written reviews (in text)	A written viewer where customers can write their experience to the product.
Community forum	A forum like feature offered by the store where people can discuss things related to the store/product.
Support ticket	A system where customers can inquiry for support questions related to the product offered by the store.
Promotion/marketing	The store offers a way to provide promotional/marketing feature to the products in the app store (e.g., featured apps, top downloads of the month).

**:* Categorical values are mutually exclusive; one and only one categorical value in the dimension can apply to a given store.

» *Monetization* — describes what, if any, payment options are provided to the user directly by the store. If a product is offered free within the store, but requires an activation key obtained elsewhere, we consider that the product is free. While most of the options are self-explanatory, some may be less obvious. For example, GitHub Marketplace offers *seat-based subscriptions* where app pricing is calculated by the number of installations made to individual machines; usually, this occurs within the context of enterprise purchase. Also, AWS offers *resource-based subscription* where the price charged is determined by the amount of resources — such as cloud storage and CPU time — that are used during the execution of the service.

» *Rights Management* — describes the Digital Rights Management (DRM) policy of the store; the values describe whether the store uses a store-wide DRM feature. For example, for Steam, all games have DRM encryption, whereas the F-Droid store contains only open source apps, so there is no need for DRM.

» *Do I need an account?* — describes whether a user can access and use the store without being registered with the app store. We find that most stores are either *account required* (e.g., Apple's App Store) or *no registration possible* (e.g., Snapcraft). However, we also found that some stores can be used without an account for some purposes, with other features requiring explicit registration; for example, the Microsoft Store allows users to download free applications without an account, but to purchase an app or leave a review, an account is required.

» *Product type* — describes the kinds of applications that are offered by the store. For example, Google Play and Steam focus on *standalone apps*, the VSCode Marketplace store offers *add-ons* to an existing tool, and AWS allows users to "rent" web-based *resources and services*.

» *Target audience* — describes the intended user base of the store. *General-purpose* stores offer products aimed at the broad general public of everyday technology users; this includes stores such as Google Play, Steam, and the Chrome Web Store. *Domain-specific* stores, on the other hand, have a dedicated focus on a specialized field; for example, Adobe Magento focuses on building e-commerce platforms.

» *Type of product creators* — describes the typical focus of creators submitting applications to the store. We distinguish between two groups of creators: those with a commercial or business focus, and those with community focus such as open source developers.

» *Intent of app store* — describes the perceived high-level goals of the app store. The values are derived from the app stores' own descriptions of their goals, often found in

"About us" web pages. For example, both F-Droid and ApkPure are *Android* app stores; however, F-Droid's focus is to provide a location to download and support FOSS software, while ApkPure's goal is to provide a location for users to be able to download *Android* apps when Google Play may be unavailable.

» *Role of intermediary* — describes the roles that the app store plays in mediating between the users and creators; these are software engineering-related services that are mostly independent of each other. For example, *checks at run time* tracks if the app store ensures that its products function correctly (e.g., Steam tracking game stats). Also, *CI/CD* indicates that the app store provides explicit support for continuous integration and deployment of the apps, which may be linked to specific development tools used by the creator.

» *Composability* — describes the relationship between products offered by the store. App stores of *independent* composability offer products that have no relationship with each other, such as Firefox Add-ons. *Vendor internal add-on/extension/unlock* means that the products within the app store can be based on each other, but only when they are from the same vendor, such as game DLC and micro-transaction unlocks. *Package managers* contain apps that can have complicated dependency relationships regardless of the creator of the products, such as the *Ubuntu* package management tool apt.

» *Analytics* — describes what kind of diagnostic information is provided by the store. We distinguish between three kinds: *Sentiment and popularity ratings* offer user-based information related to store products, such as number of installs in Home Assistant. *Marketing feedback* tracks telemetry information for creators on the performance of their product, such as GitHub Marketplace tracking retention rate for their products for creators. *Product usage data* details the observed usage of the products; for example, Steam tracks the average number of hours users spend on each product.

» *Communication channels* — tracks the types of methods the store directly offers for communications between both users and creators. Since most stores offer a *product homepage* for each of their products, the app creators are largely free to put any information here. This means that if a creator wishes, they can put links to other communication methods external to the store. We do not track such information here since it is product dependent instead of store dependent. While ratings and reviews/comments are often paired together, during our exploration, we found cases where user ratings were permitted but user reviews were not; thus, we have separate values for ratings and reviews.

Stage 2: Expanded Collection of App Stores and Labeled Set of Representative Stores

In stage 1, we identified 53 store candidates. To provide the required data for our experiments, we explored these stores to identify which of the *fundamental features* of the previous stage are true for each store. Two individuals carried out the qualitative studies in this stage, unless stated otherwise. The query results are summarized in Table 6.3. In Table 6.3, we list the search term construction keywords and the first 3 occurrence of stores by the search term. For example, in search term constructed from `(addon OR add-on)` and `("market place") OR marketplace`, the first 3 occurrences are Google Play, PrestaShop, and CS-Cart. To check the applicability of our dimensions and the labeling guidelines, we have measured the inter-rater agreement between on the 12 stores. We used the Cohen's Kappa [32] as a measurement for our inter-rater agreement. The Cohen's Kappa is widely used in software engineering research [107]. We have reached an agreement of 86.3% with Cohen's Kappa [32] of 0.711). Our agreement based on the Cohen's Kappa is considered as a substantial [77] inter-rater agreement suggesting a high confidence of agreement between the two raters.

There are many app stores beyond Google Play and Apple's App Store. These app stores exhibit a diverse set of features.

RQ2: *Are there groups of stores that share similar features?*

Using the labeled data of the 53 stores, we were able to perform the *K-means* cluster analysis that we have introduced in Section 6.3.2. With the number of clusters guided by the Silhouette method to choose the best k value for *K-means*, our clustering resulted in eight clusters.

Due to the nature of unsupervised methods, *K-means* is able to identify only the clusters and their members; no real-world meanings are extracted for why the cluster members belong together. It is also important to note that the *K-means* algorithm performs *hard clustering*; that is, it creates a partitioning of the stores into mutually exclusive groups that together span the whole space. Thus each store will be assigned to the unique cluster that the algorithm considers to best represent it. For this reason, the raw results from *K-means* should not be seen as authoritative, but rather as a vehicle for identifying groups of stores with similar characteristics. Therefore, we leverage the *K-means* clustering and further

Table 6.3: First three identified stores for each *Google* query

	`("app store" OR store)`	`("market place" OR marketplace)`	`shop`	`repository`	manager
`app`	Apple App Store, Google Play	BigCommerce, Google Play, HubSpot	Apple App Store	F-Droid, Guardian Project, IzzyOnDroid	Google Play
`software`	Mac App Store	MarketPlaceKit, Sellacious, CS-Cart	ϕ	ϕ	ϕ
`(addon OR add-on)`	Mac App Store, Home Assistant, Firefox Add-ons	Google Play, PrestaShop, CS-Cart	PrestaShop, Chrome Web Store	Kodi	CurseForge, Ajour, Minion
`(plugin OR plug-in)`	Google Play, SketchUcation, RICOH THETA	WordPress, JetBrains	Bukkit, Plugin Boutique	WordPress, JetBrains	Jenkins, JMeter, Autodesk
`(extension -hair -lash)`	Chrome Web Store, Microsoft Edge	VSCode Marketplace, Adobe Magento, Chrome Web Store	Chrome Web Store	TYPO3, GNOME SHELL	Chrome Web Store
`install`	Google Play, Apple App Store	Google Play, Eclipse	Apple App Store, Google Play, Microsoft Store	Kodi, Home Assistant, DockerHub	Google Play, APKPure, Daz3D
`solution`	Mac App Store, Microsoft Store	CS-Cart	ϕ	ϕ	ϕ
`(software library -book)`	Microsoft Store	VSCode Marketplace, QT Extensions, GitHub Marketplace	ϕ	ϕ	ϕ
`package`	Apple App Store, Google Play, Snapcraft	CS-Cart, concrete5	Google Play	Packagist, PyPI, Ubuntu Packages	Chocolatey, NPM, NuGet

examine the clusters in detail to try to derive a human understandable categorization of the stores.

We start by analyzing the differences between clusters by analyzing the definitive characteristics of each cluster. To identify the features that best characterize each cluster, we have calculated the deviation of each cluster *centroid* (i.e, the *center* of the cluster) from the *centroid-of-centroids* (C) over all clusters. In Table 6.4, we show the details of the top 10 features that deviates the most from the C. Column C contains the the *centroid-of-centroids* with values for each feature. The remaining columns represent each cluster by an index from 1 to 8. The values in these columns represent the proportion of app stores in the cluster with a specific feature, the mean, and the background color of each cell represent the deviation of the particular cluster *centroid* (i.e., difference between the centroid of this cluster and the centroid-of-centroids for the feature). Each row corresponds to a feature of the stores, which makes it easy to understand which features are descriptive of a cluster.

The table only shows the top 10 deviations per cluster (i.e., *column*) to focus on the most important contributors to each cluster. Since all features are binary (each store has or does not have the feature) all values of the centroid-of-centroids are between $[0, 1]$; thus, a positive deviation (shown with a green background) implies that the stores in the cluster are *more* likely to have the attribute, and a negative deviation (shown with a magenta background) implies that the stores are *less* likely to have the attribute.

For example, for cluster 8 the most important contributor is *"[Composability] Vendor internal add-on/extension/unlock* where the centroid of the cluster is 1. When comparing against the centroid-of-centroids (at 0.15), the deviation is at 0.85; this implies that all stores in this cluster have this feature. On the other hand, an example of negative deviation for cluster 1 is the feature *[Composability] Independent* with a centroid of 0 indicating that no stores in this cluster have this feature. Since the centroid-of-centroids for this features is at 0.56, this implies the deviation for stores in this cluster is -0.56.

After the top characteristics that make each cluster distinctive had been identified, we leveraged this information to name and describe each cluster accordingly. Using the information from Table 6.4 which shows the defining features of each cluster, we derived an organization of the clusters based on several dimensions. The results are described in Table 6.5.

One important dimension focuses on the type of application served by stores in the cluster. We identified three major types of applications that differentiate the clusters: *General*, where the store offers stand-alone programs that run without the need of specific software (aside from a specific operating system, e.g., Google Play, AWS, Steam); *Extensions*, where the store offers extensions to a specific program or platform e.g., VSCode Market-

Table 6.4: The 8 clusters found by the *K-means* algorithm, with top deviated features from the centroid of centroids (C)

(Each cell with a value represents one of the ten most influential features of the corresponding cluster. The number indicates the percentage of stores with the specific feature. The color encodes whether stores in that cluster are less (magenta) or more (green) likely to have the feature, compared to the centroid.)

		Cluster Index							
Features	**C**	1	2	3	4	5	6	7	8
Monetization									
Free	1.00								
One-time payment	0.35	0.00		0.00	0.00				1.00
Seat-based subscription	0.09		0.50						
Time-based subscription	0.30		0.75	0.00					0.86
Resource-based subscription	0.05								
Micro-transactions	0.11								0.86
Custom Pricing	0.01								
Rights Management									
Creator managed DRM	0.72		0.25	1.00					0.14
Store-enforced DRM	0.27		0.75						0.86
Do I need an account to use the store									
Account Required	0.33			0.00		0.75	1.00		0.86
No registration possible	0.35	1.00	0.00		0.00		0.00	1.00	
Some features require registration	0.30		1.00		1.00				
Product Type									
Standalone apps	0.42		0.00					1.00	
Extension/add-ons to apps/hardware	0.68	0.33						0.00	
Service/Resources	0.08								
Package/Library	0.17	0.89							
Target audience									
General purpose	0.33			0.00	0.83		0.00	1.00	
Domain-specific	0.67			1.00	0.17		1.00	0.00	
Type of product creators									
Business	0.67	0.22		0.00			1.00		
Community	0.67			1.00			0.11		
Intent of app store									
Community building / support	0.52		1.00		1.00	0.00	0.11		
Profit	0.38	0.00		0.00	0.00		0.78	0.00	1.00
Centralization of product delivery	0.84								
Expanding the platform	0.76							0.17	
Role of intermediary									
Embedded Advertisement API	0.16								0.71
CI/CD	0.05								
Checks at run time	0.14		0.50						
Quality/security checks	0.74					0.25			
Composability									
Independent	0.56	0.00			1.00	1.00			0.00
Vendor internal	0.15								1.00
Package manager type	0.19	1.00							
Analytics									
Sentiment and popularity ratings	0.73					0.00		0.33	
Marking feedback	0.25								
Product Usage data	0.33								
Communication channels									
Documentation (wikis, FAQs)	0.81					0.25			
Product homepage	0.97								
Star/Score/Up/Downvote rating	0.57	0.11	1.00		1.00	0.00	1.00	0.00	
Written reviews (in text)	0.47	0.00			1.00	0.00	0.89	0.00	
Community Forum	0.45			0.75		0.00			
Support Ticket	0.35								
Promotion/Marketing	0.71					0.25			

Table 6.5: List of stores and descriptions by cluster, with the example store that is closest to cluster centroid

Type	Stores in Cluster	Example Store	Cluster Description	Index
Extensions				
Commercial specialized	Adobe Magento, AutoDesk, BigCommerce, GoG, HubSpot, Plugin Boutique, Presta Shop, SketchUcation, CS-Cart	*Presta Shop* offers addons to the ecommerce solution platform.	Products in the stores are very domain specific. Creators are mostly business and their store front offers rating systems and written reviews.	6
Community specialized	Bukkit, CurseForge, DockerHub, Home Assistant, Jmeter, Kodi, Minion, VSCode Marketplace	*Kodi* add-on components offers extensions to the *Kodi* entertainment center.	These are community focused stores that offers free products to users. Stores also tailor to a specific domain.	3
Community non-specialized	Apkpure, Chrome Web Store, Eclipse Marketplace, Firefox Add-ons, Gnome, Wordpress	*Wordpress* offers free extensions for users using the wordpress platform.	Products in these stores offers extensions to the platform. Essential operations do not need registration (e.g., installing apps). Products offered in the stores face a generic audience and are independent from each other.	4
General				
Commercial	AWS, Google Play Store, Microsoft Store, Nintendo eShop, Steam, Samsung Galaxy Store, Apple App Store	*MicroSoft Store* offers applications for the windows platform.	Typical stores many people encounter everyday. They run for profit and offer vendor internal products supporting most monetization options.	8
Community	Chocolatey, F-Droid, Flatpak, Guardian Project Repository, IzzyOnDroid, Snapcraft	*F-Droid* is a free and open source software only *Android* application store.	These stores contain standalone free products only. Creators for the stores are mostly from the community and the products are majority open source.	7
Package Manager	Ajour, Jenkins, MacPorts, NPM, NuGet, Packagist, PyPI, Typo3, Ubuntu packages	*Packagist* is the main repository for *PHP* packages.	No account system is involved for these stores. Products are free and most in package style with interdependency relationships. Communication channels are also limited with ratings and reviews missing for most stores.	1
Subscription oriented	Github Marketplace, JetBrains, Qt Marketplace, concrete5 marketplace	*Github Marketplace* offers applications and actions to improve the workflow related to *git* repositories hosted on *GitHub*	Often offers subscription services and supports DRM management by the store. Products are not standalone applications and either provide service or extends a platform.	2
Other	MarketPlaceKit, RICOH THETA, Sellacious, daz3D	*Sellacious* is a ecommerce platform and provides extensions to the platform.	They do not have much communication channels offered. Rating and reviews do not exist in the stores. The stores mostly exists to distribute extensions centrally to the platform they are based on.	5

PLACE for *VSCode*, CHROME WEB STORE for *Google Chrome*; and *Package manager*, where the store offers stand-alone programs, but also manages dependency-relationships and requirements between different applications in the store e.g., NPM, MACPORTS, UBUNTU PACKAGES. Another dimension in which these clusters can be organized is whether they are Commercial (business-oriented) or Community-managed (no money is involved).

App stores are not all alike. Intuitive groupings emerge naturally from the data. Their differences can be due to the type of application they offer — standalone or extensions — and their operational model, either business- or community-oriented. We found that app stores in different groups of our clustering have different properties; it would thus seem unwise to treat them as all the same when studying them.

6.5 Discussion

In this section, we discuss our findings regarding what we consider app stores to be based on our clustering results, and we describe various research opportunities involving the influence of app stores on software engineering practices.

6.5.1 What Is an App Store?

The term *app store* became popular largely through Apple's APP STORE, which launched in 2008 along with the *iPhone 3G* [10]. Other online software stores have also appeared and have had the term applied to them. Originally, the term usually referred to stores of applications for mobile devices, but we have found that today there is ample diversity of the type of applications that app stores offer and in the features they provide to app developers and users. App Stores are also dynamic: features are continually being added, removed, and altered by store owners in response to changes in their goals and feedback from their socio-technical environments. For example, the CHROME WEB STORE initially introduced a built-in monetization option that provided a mechanism for applications to receive payments from its users; however, the store later decided to deprecate this monetization option [55] and suggested developers to switch to alternative payment-handling options.

In our work, we have employed a working definition through our inclusion/exclusion criteria for app stores to be included in our research. However, due to the complexity,

diversity, and constantly evolving nature of app stores, we have decided not to attempt a firm, prescriptive definition of the term. Instead, in the following paragraphs, we will discuss each of several aspects of app stores in detail, and hope that in the future, a more robust definition and operating model can emerge.

Common features of app stores

Although we found significant diversity among the example app stores we studied, we were able to identify a set of three common features that appear to span the space of app stores.

» *Simple installation and updates of apps* — An app store facilitates simple installation of a selected application, and can also enable simple updating. For some stores, apps are expected to run on the hardware of the client; in others, the app store provides and manages the hardware where the app runs. In both cases, the app store frees the user from worrying about the technical details of installation: mainly, if the app will be compatible with their specific hardware and software configuration; and the installation of the app and its dependencies (where applicable). Typically, app stores will also automate the installation of updates to the application, again freeing the user from worrying about if they have the latest version of the app with the newest features and bug fixes.

» *App exploration and discovery* — App Stores provide mechanisms that allow users to find apps they might want to use. In its simple form, this mechanism might be a search engine that returns a list of apps that match a given set of keywords (such as HOMEBREW, PyPI). In the labeled app stores, 73% of stores provide some kind of aggregated recommendations (e.g., advertisement and trends in WordPress), up to personal recommendations that are based on other apps the user has installed before (e.g., Apple's App Store). User feedback via reviews (present in 47% of the labeled app stores) and forums (present in 45% of the labeled app stores) can provide further information to aid other users in identifying apps of possible interest to them.

» *The app store guarantees the runtime environment* — In practice, app stores often execute within a runtime environment (RTE), such as an operating system (e.g., Google Play on *Android*) or an extensible software application (e.g., Firefox Add-ons on *Firefox*). Many app stores simply sit on top of the RTE, acting primarily as a gatekeeper for adding and deleting apps. However, some app stores are more tightly integrated with the RTE; in extreme cases, the app store can extend the RTE with the app store's own functionality and together provide an augmented RTE for the applications managed through the app store. Steam is a good example for extending the RTE with its own features. Developers can integrate with many services offered by Steam, such as an achievement

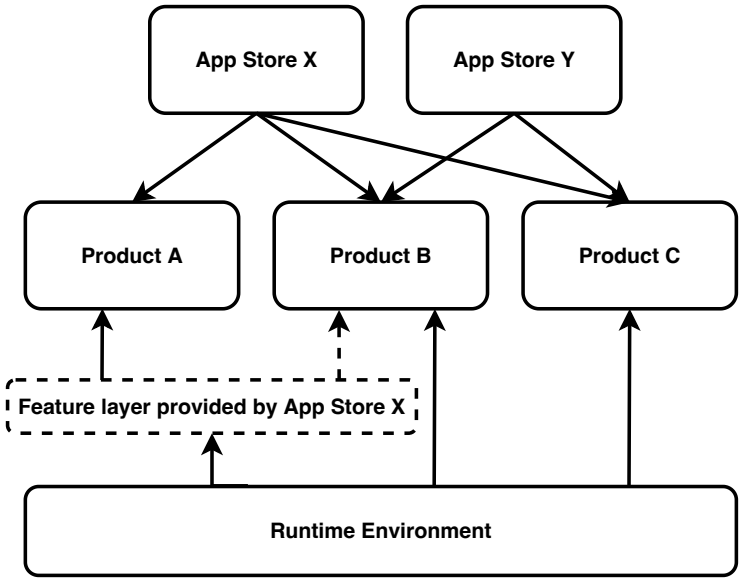


Figure 6.3: Stores may offer optional extensions to the runtime environment for applications

system that offers players recognition when they fulfill certain requirements in the game. Fig. 6.3 illustrates the situation where a product may integrate with additional store-added features to the RTE, which in turn enriches the user experience of the store users. When *Product B* is offered in *App Store Y*, it will not have the features provided by *App Store X*.

The app store ensures that apps are installed only when their runtime requirements are satisfied. The process is often done through running checks on apps submitted to the app store, which 74% of the labeled app stores perform specifically. By specifying the runtime requirements, the assumption (for both the developer and the user) is that if the application is installed (implying that the requirements are satisfied), it is expected to run properly. This is usually done by a software layer on top of the RTE (either app store or user provided). In its simplest form, this software layer is responsible for installing and updating apps (see "Simple installation and updates of apps" above). In some cases, this software layer might also include a set of libraries that the apps can use to provide features specific to the app store thus forming part of the RTE for the applications. These libraries might range in purpose (domain specific, common GUI, resource management, etc.). In extreme cases, this layer includes the operating system (as it is the case with Apple's App Store). However, checks during runtime is a very rare feature, which on 14% of the labeled app stores provides.

Some hardware platforms have become so tightly integrated to the software layer of the app store that they can be considered monolithic: the hardware is rendered unusable without the app store. This is exemplified by the Apple's App Store, where one cannot use the hardware without first having an account in the app store; even operating system upgrades are distributed via the store.

This tight level of integration has clear benefits for all three stakeholders: end users have fewer installation technical details to worry about; app developers can be assured that users will be able to install their apps without the need for technical support; and app store owners can strictly manage who has access to the user's RTE and how. However, such tight integration is technically unnecessary and may even be undesirable. From a software engineering perspective, such tight coupling could be seen as a "design smell", since the operating system and the app store layers address fundamentally different concerns. Also, tight integration can create an artificial barrier to competition, effectively establishing a quasi-monopoly for the store owner; the store owner may assume the role of gatekeeper not only for streamlining technical issues, but also for business reasons, requiring a kind of toll to be paid by app developers for access to the store. A recent initiative in the European Union [33] aims to enable fair competition by enforcing that ecosystems are opened up, which will likely also allow the installation of alternative software layers for other app stores, a term called side-loading. In contrast to the *Apple*'s tight control of the operating system as part of its app store, Android allows third-party app store software (e.g., F-Droid [46]) to be installed in co-existence with the system default (often Google Play).

As mentioned above, some stores distribute software that runs on hardware owned by the App Store itself; in these cases, the RTE is fully managed and controlled by the store. For example GitHub Marketplace and Atlassian Marketplace offer applications that run on GitHub and Atlassian servers (respectively). In most cases, these applications are not deployed to the user's computers.

Different types of app stores

While some features are broadly shared by all app stores, in Section 6.4, we identified different groups of app stores based on their features. For stores within the same group, they often share common features, and in different groups, the stores have less in common. In the following paragraphs, we discuss the differences between the groups in detail.

» *Diversity in goals* — As a platform focusing on delivering products to customers, the high-level goal of one app store can be dramatically different from the other. Even app

stores providing software for the same underlying RTE can have radically different purposes. For example, consider the app stores that run on *Android*. Google Play is the *de facto* store for *Android* applications. F-Droid store, on the other hand, offers only free and open source *Android* applications, and APKPure offers multiple versions of the same software so the user can decide which version they would like to install.

Apple's app store offers applications for all its RTEs: *MacOS* (laptop and desktops), *iOS* (phones and tables), and the *Safari* browser. In contrast, *Google* has different stores for *AndroidOS* and for its web browser, *Chrome*, (the latter no longer for profit). Microsoft Store sells hardware and apps for *Windows* and *XBox*. Alexa Skills offers skills that enhance the voice agent *Alexa*'s capabilities to do more things. Many language package systems (e.g., NPM, PyPI) are run by a different organization and extends the functionality of the core language.

In many program language ecosystems, the core language development (focusing on the language features) and packaging system (focusing on extending the functionality of the language) are led by separate organizations (e.g., NPM [100] and JavaScript [67]).

» *Diversity in business model* — Another important difference we observed is between business-managed and community-managed stores. In business-managed stores (with few exceptions), a primary goal is to generate a profit. These stores provide a payment mechanism between the app creator and the purchaser (and the store keeps a percentage of any sales). These stores have to solve three key concerns: first, implementing registration and authentication of users and developers; second, some type of digital rights management, so only users who have acquired the software can use it; and third, a payment mechanism (e.g., subscription, one-time payment, and advertisement).

Community-managed stores, on the other hand, are run by volunteers and their features focus on facilitating not-for-profit product delivery from developer to user. Many community stores offer limited community interactions compared to business stores where customer feedback is important. For example, in the Kodi store, add-ons have a web page (e.g., *The Movie Database Python* [75]). This page provides information regarding installation of the add-on, e.g., compatibility, download links, and installation requirements. Meanwhile, most communication channels about the add-on are hosted elsewhere; for example, installation and usage instructions, extended descriptions, and screenshots can be found in the community forum instead.

It is important to note that the products contained in community-oriented stores are not limited to open source software; some community-managed app store policies often permit the distribution of proprietary software. In the natural groupings we observed, no rights management are enforced from the store side for Cluster 3, meanwhile, most stores

in Cluster 8 have some form of rights management built-in to the store. For example, HOMEBREW permits apps that are not open source if the apps are free to use; these apps might include in-app purchases (such as an upgrade to a full-feature app) that are handled outside of HOMEBREW.

6.5.2 Research Opportunities Involving App Stores

App Stores are becoming the primary channel for software delivery and exert considerable influence in many aspects of the software development process. A previous study by Rosen and Shihab [119] on Stack Overflow questions by mobile developers has shown that app delivery is one of the biggest challenges developers faced. Our results in Section 6.4 demonstrate that there is a wide variety of types of stores, each with different features and goals. Today, app stores encompass many kinds of applications, from games running on the hardware of the user to add-ons for applications that run on corporate servers such as *GitHub*. However, existing research often focuses heavily on the applications offered inside app stores, especially those of the two major mobile app stores. In the following paragraphs, we discuss several research opportunities to study how app stores can affect software development.

App Stores as actors in software development

» *App Stores affect the software product cycle* — Researchers need to consider how and why app stores can affect the software development life cycle. For example, we know that app stores can constrain and sometimes even dictate software release processes. Some stores go beyond this and exert a kind of socio-technical environmental pressure on other software development practices, becoming a de facto stakeholder in app development. Sometimes these environmental pressures are technical in nature, where the app store might dictate the programming language or deployment platform/OS; some app stores go further and create RTEs, software development kits (SDKs), and user interface (UI) libraries that must be used by all app developers. Sometimes these environmental pressures are non-technical in nature, such as when the app store prescribes the kinds of application that is allowed in the store. For example, *Microsoft* recently announced that it will not permit app developers to profit from open source applications.[2] When an app store operates in a manner such that it has control over what kind of application to include, it creates a software ecosystem

[2]See Update to 10.8.7 https://docs.microsoft.com/en-us/windows/uwp/publish/store-policies-change-history

and as such, it faces the same challenges that any other ecosystem has: how to thrive. In particular, stores need to understand the needs of their developers and users to retain existing ones and attract new ones. However, suggested by what we have observed in Section 6.4, app stores are diverse with a large number of features that characterize (and differentiate) them. While stores are experimenting and evolving, each action is likely to have an effect on the ecosystems they formed (both positively and negatively). Thus, the impact of app stores in the economy and their markets is worthy of further study.

» *An app may be offered in several app stores* — Developers want to run their software on the platform that is provided or supported by the store, and as such they must accept the requirements and limitations that such a store may impose. This issue is compounded when the app is being offered in more than one store, as the developers might have to adapt their processes to different sets of requirements, some of which might be conflicting. For instance, an app can be both available in F-Droid (in Cluster 7) and Google Play (in Cluster 8). In Google Play, it is common for applications to collect telemetry data to better understand typical user behaviour; however, in F-Droid — an open source and privacy-oriented store — such data collection is highly discouraged. Furthermore, developers must also adapt to the features and limitations that a store provides regarding software deployment, communication with users and — when they exist — the mechanism available to profit from their software and to use digital rights management. This is particularly interesting if the targeted app stores are in different natural groupings. This introduces new areas of studies such as how store policies propagate to applications over time, and how violations of store policies can be detected automatically. Researchers have already begun to investigate this topic through qualitative approaches to identify how applications comply with specific policies that concern accessibility [7] and human values [101].

» *App stores strongly affect the release engineering process* — App Stores are especially important in release engineering. Specifically, the release process needs to consider how the application is to be packaged, deployed, and updated. The heterogeneity of the platform provided by RTEs might also affect the number of versions of the application that need to be deployed (e.g., variety of target CPUs, screen sizes, screen orientations, and memory available).

When an application is developed for multiple stores, it must effectively be managed as a product line; this is because multiple deliverables must be created, one for each platform-store combination [154]. Multiple deliverables can also help for telemetry reasons such as tracking the installation source of the application [98]. The differences between packaged versions might be as significant as requiring the source code to be written in different programming languages, using different frameworks; also, each store is likely to require

different deployment processes.

For example, when cross-releasing browser add-ons, developers may have to rewrite part of the functionality in *Swift/Objective-C* for better integration with Safari (in the Apple's App Store), while at the same time maintaining a fully *JavaScript* version for Chrome Web Store. Also, the scheduling of release activities is often dictated by the release processes of the stores. A previous study has showed that taking into consideration of app review times is an important factor when planning releases [3]. The app store standardizes, and often simplifies, the release engineering processes for its store; but it also becomes a potential roadblock that might delay (or even reject) a new release.

The challenge of transferring understanding between stores

As noted above, prior work has examined many aspects of app stores, yet the app store itself has rarely been the focus of the research. In many studies, the app store serves as a convenient collection of apps, and the research focuses on mobile development concerns such as testing and bug localization. Even when research focuses on the app store itself, the scope rarely extends beyond Google Play and Apple's App Store. Based on our observations, the diversity of app stores in their operational goals, business models, delivery channels, and feature sets can affect the generalizability of research outcomes. For example, there have recently been many studies [35, 80, 165, 59, 58, 101, 48] that focus on app reviews. However, for an app store that does not have reviews (e.g., Nintendo eShop) none of the findings and tools can be leveraged (e.g., stores in Cluster 1, 5, and 7).

» *App Stores that have the same features may still differ significantly* — Depending on the problem domain, the details of software development practices can vary dramatically. For example, game development has been compared to both more traditional industrial software development [93] and to open source software development [106]; in both cases, the development processes can differ greatly. We conjecture that the same may also occur across app stores, where despite the same feature is being offered in the different stores, the convention of using them could be different. As mentioned above, one specific observation has been made between the gaming-focused store Steam and mobile stores (e.g., Google Play) in Cluster 8, where Lin et al. [80] found that reviews across the platforms for the same app were often quite different in tone. Such uncertainly invites future research to validate their findings in one store to another to improve the generalizability of the results, and also encourages replication studies to verify existing results on other stores.

» *A feature not in the app store does not mean the functionality is missing* — While some app stores aim to provide a complete experience, where all interactions from the developers

and users are expected to be performed with in the store, some app stores export part of the work to other platforms. This can even occur for common features that one might find essential. For instance, starred reviews is universal in Cluster 2, 4, and 6 where typical users leverage this information to decide whether an application is good, starred reviews do not or rarely exists for other stores in Cluster 1, 5, 7. The specialized store may have some other metric to indicate popularity or quality, such as total number of downloads, but the focus of the store is often to offer a managed way of installation. Other features, such as application support, are left to other platforms (e.g., social media). Research can further explore the integration between app stores and other platforms.

6.6 Threats to Validity

» *Internal Validity* — We create our initial seeding of app stores from personal experience. Personal bias could cause us to miss other types of app stores. However, given the number of individuals involved in this study and our initial effort to consider as many stores as possible, we feel that have created a wide, deep, and collaborative "best effort". When we labeled app stores by their dimensions, it is a qualitative process. As with any qualitative process, the results could be biased. We tackled this issue by first labeling a few stores separately and discussing the results until a consensus was achieved; thus, we started with a set of "gold standard" labels. Then the labeling task was delegated to two individuals who continued to label the stores separately with a portion of the store overlapping. The overlapping labels are then verified by the *Cohen's Kappa* between the two individuals to measure the agreement.

We leveraged the *K-means* algorithm for the clustering process. We first applied *PCA* techniques to reduce the dimensions of the initial labeling and provide an orthogonal basis to feed the *K-means* clustering. When using other clustering algorithms (e.g., *Mean-shift*, *DBSCAN*), the clustering result might change; while *K-means* is widely adopted for clustering process in SE research, by nature, determining the proper k value is still a challenge. We followed common best practice to use metrics (i.e., the *Silhouette* method) to determine the best value k. Despite our efforts, the output of the *K-means* clustering is not perfect. We mainly leveraged the *K-means* clustering as the first step to illustrate that app stores forms natural clusters which are different from each other. Base on the *K-means* output, we further grouped the clusters into types based on our qualitative understanding of the app store space.

» *External Validity* — During the process of expanding app stores, we relied on the *Google Search Engine* to find web results based on keywords. The results of this step rely on

the capability of *Google* and are subject to change over time as *Google* updates its search algorithms. The order may also be affected by SEO operations. Combining results from other search engines (e.g., *DuckDuckGo*, *Bing*) can help to reduce the bias.

When we applied our inclusion criteria, 1) app stores must contain software products and 2) should offer an end-to-end experience for users (ordering, delivery, installation), we excluded stores that focus on digital assets that are not software, such as a pure assets store that offers cosmetic enhancements to desktop environments; we also excluded stores that offer software products but in a way such that installation is completely managed by users. An extreme example, will be the software section of *Amazon* where software are sold as activation keys where the users would input to activate the software which they needs to install themselves. A more general inspection of all means of distribution software can be performed to gain a broader understanding of software distribution.

We relied on only publicly available information to label each store. So if some functionality (e.g., analytics information) is not documented publicly, we were unable to confirm whether the store has such functionality. We also set a time limit to label each store so in case we were unable to find information about the store, with each store receives the same amount of attention.

One of the main challenges for reproducibility and replicability is that the *Google Search* results and app stores can change overtime. So in the future, if researchers would like to repeat our study, the labeling results may differ due to updates in the app store. To mitigate this issue, we've included a snapshot of all *Google Search* results, and documented how we would perform the labeling. So while the final labels may differ, by applying the same process, a replication study would be possible with updated data.

6.7 Summary

In this chapter, we explore on the idea of what an app store is and what features make app stores unique from each other. We labeled a set of representative stores, curated from web search queries, by their features to study the natural groupings of the stores. Our analysis suggests that app stores can differ in the type of product offered in the store, and whether the store is business oriented or community oriented. These natural groupings of the stores challenge the manner in which app store research has largely been mobile focused. Previous studies have already shown empirical differences in activities in mobile app stores and game stores [80]. Our study further suggests that in the future, when we study app stores, we would need to consider the generalizability of the results across app

stores. Since one type of app store may operate under different constraints than another kind, results observed in one app store setting may not generalize to others.

Chapter 7

Conclusions

Software engineering focuses on delivering software products that solve real-world problems. Successfully delivering a software product requires many resources, such as external libraries, issue tracking systems for debugging, etc. Furthermore, recent advancements encourage in software development have encouraged an agile approach, which involves releasing software products in cycles and continuously improving them until they reach a desirable state. To achieve this, software developers must work collaboratively and be knowledgable in various aspects of software development, such as design, implementation, testing, and maintenance.

The social aspect of software development means that there are frequent interactions between developers and users throughout the development process. These interactions can involve estimating the performance of the software or gathering information related to software development, and they leave traces in the form of software artifacts. Examples of software artifacts include version control information, technical documentation, runtime log, etc. These software artifacts contain valuable information that record every aspect of the software product, from the planning phase of designing the software product to the implementation phase of developing the software. By analyzing these artifacts, developers can gain insights into technical design decisions, software quality, and user perception of the software product. This information can be used to improve the software product and ensure that it meets the needs of users.

In Chapter 4, we studied artifacts containing Q&A sessions on Stack Overflow and discovered that the technical Q&A is often more complex than simply providing direct answers to questions. Many questions involve discussions before a satisfactory answer is found. This presents a challenge for preserving Q&A knowledge, as interactive environments like chat

rooms are ideal for Q&A, but not for long-term viewing [115]. Our observations highlight the need for auto summarization tools that can effectively capture and preserve the information contained in Q&A sessions for future reference. By developing such tools, we can ensure that valuable knowledge is not lost and can be easily accessed by developers in the future.

In Chapter 5, we studied publicly shared user-specific configuration files and found that developers often dedicate significant effort to maintaining both the *dotfiles* repository and the *dotfiles*. This suggests that there may be room for a more standardized method of managing configuration files in user space. Additionally, the corpus of real-world configuration files can provide valuable insights for tool developers, helping them with understanding how their tools are used in practice and improving them accordingly.

In Chapter 6, we investigate the role of app stores in software distribution and found that app store have become the intermediary between the developers and users. By providing a platform that connects developers with users, app stores have created a bridge that facilitates the distribution of software. Existing research have a primarily focus on mobile app stores, but our findings suggest that a variety of app store exist, offering different functionalities and serving unique purposes. For example, there are app stores specialize in distributing software extensions, while other focus on distributing subscriptions to software services.

In the remainder of the chapter, we outline the contributions of this thesis and explore future research directions.

7.1 Summary of Contributions

Overall, we explored three different crowd-based software repositories created from distinct models of developer-user interactions. Despite the difference, the studied software artifacts are often created by one group of users and beneficial to another group. However, in our thesis, we find that challenges can arise in both creating and maintaining the information contained in the software artifacts.

Thesis Statement: *Studying crowd-based software repositories can enhance our understanding of developer-user interactions during the creation of software artifacts. With this deeper understanding, we can identify challenges in software development and provide valuable insights to improve the software development process.*

7.1.1 Enhanced Understanding of Developer-User Interactions

In Chapter 4, we study the user interactions in comments associated with questions on Stack Overflow. Serving as one of the most essential places for knowledge acquisition, the quality of content on Stack Overflow is important. Our study highlights that the creation of answers on the platform is often not a direct process, the question discussions facilitates the process. Specifically, our contributions are:

- We find that a significant amount of questions on Stack Overflow have associated comments.
- We observe that when question discussion occur, the majority of the discussion began before any answer.
- We show that askers and answerers actively participate in question discussions.
- We evaluate the effects of question discussions on facilitating the question answering process.

In Chapter 5, we investigate user-specific configuration files shared by developers on *GitHub*. As a complicated task, software development rely on a plethora of tools to aid the process. These tools are often highly customizable, and can be adapted to fit different tasks. *dotfiles* repositories are a collection of the configuration files to customize the software tools. Our contributions to understanding *dotfiles* repositories include:

- We show that sharing *dotfiles* is a common practice among prolific *GitHub* users.
- We provide a taxonomy based on common *dotfiles*. The most common *dotfiles* are related to shells, text editors, and *dotfiles* management.
- We study the intent of *dotfiles* updates through analyzing commits in *dotfiles* repositories. We observe that while most commits are related to tweaking the configuration files, many commits focus on *dotfiles* management and documentation.
- We propose a method to model file change history as a time-series based on code churn history. Based on the time-series model, we illustrate the code churn history patterns of updating *dotfiles*, showcasing the evolution of *dotfiles*.

In Chapter 6, we study the concept of app stores in software development. As the target distribution platform for many software products, our current understanding of app stores are largely based on the two mobile platforms. The main contributions of this study is as follows:

- We provide a working definition of app stores base on existing understanding that captures both its functionality and its role in software distribution.
- We identify a set of features that can describe app stores.
- We provide a labeled set of representative app stores based on the feature derived.
- We show that based on the features of app stores, a natural grouping exists, suggesting a diversity in the operation of app stores.
- We discuss insights on the diversity of app stores and how app stores affects the software engineering process.

7.1.2 Identified Challenges

With the enhanced understanding gained through studying the three models of developer-user interactions, we identified four challenges.

- We need to incorporate the users' needs in technical Q&A. Despite the improvement Stack Overflow provides over traditional Q&A methods, in practice, canonical answers to canonical questions is not always the case. Users still require some form of interactions to facilitate the answering process.
- There is room for improvement in distributed knowledge from Q&A sessions. Given our enhanced understanding, viewers of question threads on Stack Overflow may need to go through the entire question discussion and answer to find the specific information that they seek. We can apply new advancements in the field to automate information summary and reduce the effort required to find information.
- Managing user-specific configurations requires additional setup. From our observations in common files in *dotfiles* repositories and the reasons why *dotfiles* repositories are updated, we find a large presence of files and updates that are related to the management of *dotfiles* instead of the configurations themselves. Moreover, developers follow different practices to manage *dotfiles*. This adds complexity for other developers who wish to learn from the *dotfiles*.

- The Apple's App Store and Google Play are not the only app stores. With our exploratory study on the spectral of app stores, we find 8 natural groupings of app stores. Both the mobile app stores belong to the same grouping, suggesting that our existing understanding of app stores may not generalize. Moreover, existing research have already shown differences between Steam and Google Play.

7.2 Avenues for Future Research

With the introduction of new technology and new development practices, new software repositories will be formed based on the by-product software artifacts. We envision a forever challenge in this area to understand new and emerging software repositories, and also leveraging existing software repositories. Specifically, based on our findings, we present the following possible directions for future research.

» *Software engineering specific language models for creating knowledge bases.* — One of the main challenges of knowledge sharing is to to present the relevant information to the users. This challenge can be repeated find across software repositories. While recent success on language models such as GPT3.0 [20] has been shown to be successful in providing information, we believe that by considering the domain specific information in software engineering, we can provide enhanced user experiences to software developers.

Specifically, we believe that with specialized summary tools, we can provide a universal platform for Q&A type of knowledge sharing. An ongoing challenge in Q&A is how to document the result of a Q&A session. As shown in our study on question discussions (Chapter 4), the need for interaction is essential to answer many questions. However, if only presented by the question and answer, valuable information from the interactions that lead to the answer could be overlooked. With next generation language models, an automatic summary can be created and presented for developers. Along with the sources, the developers can further trace the information if needed.

Another useful perspective is to create cookbook style documentation using real-world use cases. The *dotfiles* repositories offers valuable information on tool configurations. However, for developers to learn from other developers *dotfiles*, it requires the developers to have an overall understanding of the software tool. A domain specific language model can extract information form the existing *dotfiles* repositories, and create cookbook style documentation for the software tools.

» *Analyzing software usage information to guide software design.* — Receiving user feedback is a hard task in software development. Even if feedback is available, it might be

biased (e.g., vocal majority, biased sampling). It is especially hard for open source software tools since any form of telemetry is often disliked by the community [97]. Future studies can explore on the idea of leveraging a collection of shared information online to better understand software usage. Specifically, we believe through analyzing specific configurations in *dotfiles*, we can understand usage patterns of the software. The usage pattern will complement user discussion and can serve as a basis for evidence on software design decisions and help guide the creation of sane defaults.

Analyzing app store artifacts is also an area that is worth exploring. For example, existing studies have already leveraged app reviews [35, 165] to discover user requirements in software development.

References

[1] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. In *Empirical Software Engineering*. Springer, 2011.

[2] Saeed Aghabozorgi, Ali Seyed Shirkhorshidi, and Teh Ying Wah. Time-series clustering–a decade review. *Information systems*, 2015.

[3] Afnan A. Al-Subaihin, Federica Sarro, Sue Black, Licia Capra, and Mark Harman. App Store Effects on Software Engineering Practices. In *Transactions on Software Engineering*. IEEE, 2021.

[4] Rana Alkadhi, Teodora Lata, Emitza Guzmany, and Bernd Bruegge. Rationale in development chat messages: an exploratory study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017.

[5] Miltiadis Allamanis and Charles Sutton. Why, when, and what: analyzing stack overflow questions by topic, type, and code. In *2013 10th Working Conference on Mining Software Repositories*, MSR '13, pages 53–56. IEEE, 2013.

[6] Sumaya Almanee, Arda Ünal, Mathias Payer, and Joshua Garcia. Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code. In *Int. Conf. on Software Engineering*. IEEE, 2021.

[7] Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. Accessibility issues in android apps: state of affairs, sentiments, and ways forward. In *Int. Conf. on Software Engineering*. IEEE, 2020.

[8] Amazon. AWS Marketplace: Homepage. https://aws.amazon.com/marketplace/, 2022. Accessed: Jun. 22, 2022.

[9] Ashton Anderson, Daniel Huttenlocher, Jon Kleinberg, and Jure Leskovec. Discovering value from community activity on focused question answering sites: a case study of Stack Overflow. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '12, 2012.

[10] Apple. Apple Introduces the New iPhone 3G. https://www.apple.com/ca/newsroom/2008/06/09Apple-Introduces-the-New-iPhone-3G/, 2008. Accessed: Jul. 17, 2022.

[11] Steven Arzt. Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection. In *Int. Conf. on Software Engineering*. IEEE, 2021.

[12] Muhammad Asaduzzaman, Ahmed Shah Mashiyat, Chanchal K Roy, and Kevin A Schneider. Answering questions about unanswered questions of stack overflow. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013.

[13] Autodesk. Autodesk App Store : Plugins, Add-ons for Autodesk software, AutoCAD, Revit, Inventor, 3ds Max, Maya ... https://apps.autodesk.com/, 2022. Accessed: Jun. 22, 2022.

[14] Sebastian Baltes, Lorik Dumani, Christoph Treude, and Stephan Diehl. SOTorrent: reconstructing and analyzing the evolution of Stack Overflow posts. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, 2018.

[15] Olga Baysal, Reid Holmes, and Michael W Godfrey. Situational awareness: personalizing issue tracking systems. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[16] Dane Bertram, Amy Voida, Saul Greenberg, and Robert Walker. Communication, Collaboration, and Bugs: The Social Nature of Issue Tracking in Small, Collocated Teams. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, CSCW '10. Association for Computing Machinery, 2010.

[17] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008.

[18] Stefanie Beyer and Martin Pinzger. Synonym suggestion for tags on stack overflow. In *Proceedings of the 23rd International Conference on Program Comprehension*, ICPC '15. IEEE, 2015.

[19] Tegawendé F Bissyandé, David Lo, Lingxiao Jiang, Laurent Réveillere, Jacques Klein, and Yves Le Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*. IEEE, 2013.

[20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 2020.

[21] Liang Cai, Haoye Wang, Bowen Xu, Qiao Huang, Xin Xia, David Lo, and Zhenchang Xing. Answerbot: an answer summary generation tool based on stack overflow. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, FSE '19, pages 1134–1138, 2019.

[22] Fabio Calefato, Filippo Lanubile, and Nicole Novielli. How to ask for technical help? Evidence-based guidelines for writing questions on Stack Overflow. *Information and Software Technology*, 2018.

[23] Canonical. Ubuntu Software Center in Launchpad. https://launchpad.net/software-center, 2009. Accessed: Jun. 22, 2022.

[24] Preetha Chatterjee, Kostadin Damevski, Lori Pollock, Vinay Augustine, and Nicholas A Kraft. Exploratory study of slack q&a chats as a mining source for software engineering tools. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019.

[25] Chunyang Chen, Xi Chen, Jiamou Sun, Zhenchang Xing, and Guoqiang Li. Data-Driven Proactive Policy Assurance of Post Quality in Community Q&A Sites. *Proceedings of the 2018 ACM Human-Computer Interaction*, 2018.

[26] Jiachi Chen, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. Maintenance-related concerns for post-deployed Ethereum smart contract development: issues, techniques, and future challenges. In *Empirical Software Engineering*. Springer, 2021.

[27] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhut, Guoqiang Li, and Jinshui Wang. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *Int. Conf. on Software Engineering*. IEEE, 2020.

[28] Mengsu Chen, Felix Fischer, Na Meng, Xiaoyin Wang, and Jens Grossklags. How reliable is the crowdsourced knowledge of security implementation? In *2019 IEEE/ACM 41st International Conference on Software Engineering*, ICSE '19, pages 536–547. IEEE, 2019.

[29] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. An empirical assessment of security risks of global android banking apps. In *Int. Conf. on Software Engineering*. IEEE, 2020.

[30] Morakot Choetkiertikul, Daniel Avery, Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. Who will answer my question on stack overflow? In *2015 24th Australasian software engineering conference*. IEEE, 2015.

[31] Shaiful Alam Chowdhury and Abram Hindle. Mining StackOverflow to Filter out Off-topic IRC Discussion. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, 2015.

[32] Jacob Cohen. A coefficient of agreement for nominal scales. In *Educational and psychological measurement*. Sage, 1960.

[33] Europeon Commission. Digital Markets Act: Commission welcomes political agreement on rules to ensure fair and open digital markets. https://ec.europa.eu/commission/presscorner/detail/en/IP_22_1978, 2022. Accessed: Jul. 13, 2022.

[34] Anthony Peter Macmillan Coxon et al. *Sorting data: Collection and analysis*. Sage, 1999.

[35] Jacek Dąbrowski, Emmanuel Letier, Anna Perini, and Angelo Susi. Analysing app reviews for software engineering: a systematic literature review. In *Empirical Software Engineering*. Springer, 2022.

[36] Kostadin Damevski, David C Shepherd, Johannes Schneider, and Lori Pollock. Mining sequences of developer interactions in visual studio for usage smells. *IEEE Transactions on Software Engineering*, 2016.

[37] Andrea Di Sorbo, Sebastiano Panichella, Corrado A Visaggio, Massimiliano Di Penta, Gerardo Canfora, and Harald C Gall. Exploiting natural language structures in software informal documentation. *IEEE Transactions on Software Engineering*, 2019.

[38] Yvonne Dittrich and Rosalba Giuffrida. Exploring the role of instant messaging in a global software development project. In *2011 IEEE Sixth International Conference on Global Software Engineering*. IEEE, 2011.

[39] Colin Dixon, Ratul Mahajan, Sharad Agarwal, AJ Brush, Bongshin Lee, Stefan Saroiu, and Victor Bahl. The home needs an operating system (and an app store). In *SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010.

[40] Docker. Explore Docker's Container Image Repository | Docker Hub. https://hub.docker.com/search?q=, 2022. Accessed: Jun. 22, 2022.

[41] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-travel testing of android apps. In *Int. Conf. on Software Engineering*. IEEE, 2020.

[42] .dotfiles. GitHub does dotfiles - dotfiles.github.io. https://dotfiles.github.io/, 2022. Accessed: Aug. 16, 2022.

[43] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proc. VLDB Endow.*, Aug 2009.

[44] James L Elshoff and Michael Marcotty. Improving computer program readability to aid modification. *Communications of the ACM*, 1982.

[45] Stack Exchange. Stack exchange data dump. https://archive.org/details/stackexchange. Accessed: 2021-12-29.

[46] F-Droid. F-Droid - Free and Open Source Android App Repository. https://f-droid.org/, 2022. Accessed: Oct. 02, 2022.

[47] Runhan Feng, Ziyang Yan, Shiyan Peng, and Yuanyuan Zhang. Automated detection of password leakage from public github repositories. In *International Conference on Software Engineering (ICSE'22)*, 2022.

[48] Ricarda Anna-Lena Fischer, Rita Walczuch, and Emitza Guzman. Does culture matter? impact of individualism and uncertainty avoidance on app reviews. In *Int. Conf. on Software Engineering: Software Engineering in Society*. IEEE, 2021.

[49] Denae Ford, Kristina Lustig, Jeremy Banks, and Chris Parnin. "We Don't Do That Here": How collaborative editing with mentors improves engagement in social Q&A communities. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18. Association for Computing Machinery, 2018.

[50] Free Software Foundation. Stow - GNU Project -Free Software Foundation. https://gnu.org/software/stow/, 2016. Accessed: Mar. 1, 2023.

[51] Neelamadhav Gantayat, Pankaj Dhoolia, Rohan Padhye, Senthil Mani, and Vibha Singhal Sinha. The synergy between voting and acceptance of answers on stackoverflow-or the lack thereof. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015.

[52] GitHub. GitHub Copilot · Your AI pair programmer. https://copilot.github.com/, 2021. Accessed: Nov. 04, 2021.

[53] GitHub. GitHub Marketplace · to improve your workflow · GitHub. https://github.com/marketplace?type=, 2022. Accessed: Jun. 06 2022.

[54] Google. Chrome Web Store - Extensions. https://chrome.google.com/webstore/category/extensions, 2022. Accessed: Jun. 22, 2022.

[55] Google. Chrome Web Store payments deprecation. https://developer.chrome.com/docs/webstore/cws-payments-deprecation/, 2022. Accessed: Mar. 16, 2022.

[56] Tomasz Górecki and Paweł Piasecki. A comprehensive comparison of distance measures for time series classification. In *Workshop on Stochastic Models, Statistics and their Application*, pages 409–428. Springer, 2019.

[57] Georgios Gousios and Diomidis Spinellis. GHTorrent: GitHub's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012.

[58] Hui Guo and Munindar P Singh. Caspar: extracting and synthesizing user stories of problems from app reviews. In *Int. Conf. on Software Engineering*. IEEE, 2020.

[59] Marlo Haering, Christoph Stanik, and Walid Maalej. Automatically matching bug reports with related app reviews. In *Int. Conf. on Software Engineering*. IEEE, 2021.

[60] Omar Haggag, Sherif Haggag, John Grundy, and Mohamed Abdelrazek. COVID-19 vs social media apps: does privacy really matter? In *Int. Conf. on Software Engineering: Software Engineering in Society*. IEEE, 2021.

[61] Mark Harman, Yue Jia, and Yuanyuan Zhang. App store mining and analysis: MSR for App Stores. In *Int. Conf. on Mining Software Repositories*. IEEE, 2012.

[62] Vincent J Hellendoorn, Sebastian Proksch, Harald C Gall, and Alberto Bacchelli. When code completion fails: A case study on real-world completions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019.

[63] Rashina Hoda, James Noble, and Stuart Marshall. Developing a grounded theory to explain the practices of self-organizing Agile teams. In *Empirical Software Engineering*. Springer, 2012.

[64] Zach Holman. Dotfiles Are Meant to Be Forked. https://zachholman.com/2010/08/dotfiles-are-meant-to-be-forked/, 2010. Accessed: Aug. 16, 2022.

[65] Daqing Hou and David M Pletcher. An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011.

[66] Yangyu Hu, Haoyu Wang, Tiantong Ji, Xusheng Xiao, Xiapu Luo, Peng Gao, and Yao Guo. Champ: Characterizing undesired app behaviors from user comments based on market policies. In *Int. Conf. on Software Engineering*. IEEE, 2021.

[67] Ecma International. TC39 - Specifying JavaScript. https://tc39.es/, 2022. Accessed: Oct. 02, 2022.

[68] X. Jin and F. Servant. What Edits are Done on the Highly Answered Questions in Stack Overflow? An Empirical Study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories*, MSR '19, 2019.

[69] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[70] Capers Jones. *Software assessments, benchmarks, and best practices*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[71] @k0kubun. Users Ranking - Gitstar Ranking. https://gitstar-ranking.com/users, 2014. Accessed: Sept. 28, 2022.

[72] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, 2014.

[73] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. Distance-based sampling of software configuration spaces. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019.

[74] David Kavaler, Asher Trockman, Bogdan Vasilescu, and Vladimir Filkov. Tool choice matters: Javascript quality assurance tools and usage outcomes in github projects. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 476–487. IEEE, 2019.

[75] Team Kodi. The Movie Database Python | Matrix | Addons | Kodi. https://kodi.tv/addons/matrix/metadata.themoviedb.org.python, 2022. Accessed: Jul. 13, 2022.

[76] Konstantin Kuznetsov, Chen Fu, Song Gao, David N Jansen, Lijun Zhang, and Andreas Zeller. Frontmatter: mining Android user interfaces at scale. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2021.

[77] Charles A Lantz and Elliott Nebenzahl. Behavior and interpretation of the κ statistic: Resolution of the two paradoxes. In *Journal of clinical epidemiology*. Elsevier, 1996.

[78] Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*. Soviet Union, 1966.

[79] Bin Lin, Alexey Zagalsky, Margaret-Anne Storey, and Alexander Serebrenik. Why developers are slacking off: Understanding how software teams use slack. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*, 2016.

[80] Dayi Lin, Cor-Paul Bezemer, Ying Zou, and Ahmed E Hassan. An empirical study of game reviews on the steam platform. In *Empirical Software Engineering*. Springer, 2019.

[81] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do api changes trigger stack overflow discussions? a study

on the android sdk. In *proceedings of the 22nd International Conference on Program Comprehension*, pages 83–94, 2014.

[82] Pei Liu, Li Li, Yichun Yan, Mattia Fazzini, and John Grundy. Identifying and characterizing silently-evolved methods in the android API. In *Int. Conf. on Software Engineering: Software Engineering in Practice*. IEEE, 2021.

[83] Siqi Ma, Juanru Li, Hyoungshick Kim, Elisa Bertino, Surya Nepal, Diethelm Ostry, and Cong Sun. Fine with "1234"? An Analysis of SMS One-Time Password Randomness in Android Apps. In *Int. Conf. on Software Engineering*. IEEE, 2021.

[84] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Oakland, CA, USA, 1967.

[85] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. In *Transactions on Software Engineering*. IEEE, 2016.

[86] Zainab Masood, Rashina Hoda, and Kelly Blincoe. How agile teams make self-assignment work: a grounded theory study. In *Empirical Software Engineering*. Springer, 2020.

[87] Andrew Meneely, Mackenzie Corcoran, and Laurie Williams. Improving developer activity metrics with issue tracking annotations. In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, 2010.

[88] Microsoft. Official page for Language Server Protocol. https://microsoft.github.io/language-server-protocol/, 2021. Accessed: Nov. 04, 2021.

[89] Microsoft. Registry. https://learn.microsoft.com/en-us/windows/win32/sysinfo/registry, 2021. Accessed: Mar. 4, 2023.

[90] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. Accurate modeling of performance histories for evolving software systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019.

[91] Vijayaraghavan Murali, Edward Yao, Umang Mathur, and Satish Chandra. Scalable statistical root cause analysis on app telemetry. In *Int. Conf. on Software Engineering: Software Engineering in Practice*. IEEE, 2021.

[92] Emerson Murphy-Hill, Edward K Smith, Caitlin Sadowski, Ciera Jaspan, Collin Winter, Matthew Jorde, Andrea Knight, Andrew Trenk, and Steve Gross. Do developers discover new tools on the toilet? In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 465–475. IEEE, 2019.

[93] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In *Int. Conf. on Software Engineering*, 2014.

[94] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Where do configuration constraints stem from? an extraction approach and an empirical study. *IEEE Transactions on Software Engineering (TSE)*, 2015.

[95] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. Finding faster configurations using flash. *IEEE Transactions on Software Engineering*, 2018.

[96] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example?: A study of programming Q&A in StackOverflow. In *Proceedings of the 28th International Conference on Software Maintenance*, ICSM '12, 2012.

[97] Hacker News. Transparent telemetry for open-source projects | hacker news. https://news.ycombinator.com/item?id=34707583, 2023. Accessed: Apr. 26, 2023.

[98] Yi Ying Ng, Hucheng Zhou, Zhiyuan Ji, Huan Luo, and Yuan Dong. Which Android app store can be trusted in China? In *Computer Software and Applications Conference*. IEEE, 2014.

[99] Tam Nguyen, Phong Vu, and Tung Nguyen. Code recommendation for exception handling. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2020.

[100] npm. npm About. https://www.npmjs.com/about, 2022. Accessed: Oct. 02, 2022.

[101] Humphrey O Obie, Waqar Hussain, Xin Xia, John Grundy, Li Li, Burak Turhan, Jon Whittle, and Mojtaba Shahin. A first look at human values-violation in app reviews. In *Int. Conf. on Software Engineering: Software Engineering in Society*. IEEE, 2021.

[102] Cyrus Omar, Young Seok Yoon, Thomas D LaToza, and Brad A Myers. Active code completion. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012.

[103] Stack Overflow. Stack Overflow Developer Survey 2022. https://survey.stackoverflow.co/2022/, 2022. Accessed: Jul. 24, 2022.

[104] Linjie Pan, Baoquan Cui, Hao Liu, Jiwei Yan, Siqi Wang, Jun Yan, and Jian Zhang. Static asynchronous component misuse detection for Android applications. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2020.

[105] John Paparrizos and Luis Gravano. k-shape: Efficient and accurate clustering of time series. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015.

[106] Luca Pascarella, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli. How is video game development different from software development in open source? In *Int. Conf. on Mining Software Repositories*. IEEE, 2018.

[107] Jorge Pérez, Jessica Díaz, Javier Garcia-Martin, and Bernardo Tabuenca. Systematic literature reviews in software engineering—Enhancement of the study selection process using Cohen's kappa statistic. In *Journal of Systems and Software*. Elsevier, 2020.

[108] Rob Pike. A lesson in shortcuts. https://archive.ph/vfXl2, 2012. Accessed: Dec. 13, 2022.

[109] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, and Michele Lanza. Understanding and classifying the quality of technical forum questions. In *2014 14th International Conference on Quality Software*. IEEE, 2014.

[110] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, Michele Lanza, and David Fullerton. Improving low quality stack overflow post detection. In *2014 IEEE international conference on software maintenance and evolution*. IEEE, 2014.

[111] Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, Giuseppe Bianco, and Rocco Oliveto. Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering*, 2019.

[112] Sydur Rahaman, Iulian Neamtiu, and Xin Yin. Algebraic-datatype taint tracking, with applications to understanding Android identifier leaks. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2021.

[113] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[114] Eric Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 1999.

[115] Eric S. Raymond. How to ask questions the smart way. http://www.catb.org/~esr/faqs/smart-questions.html. Accessed: 2019-10-21.

[116] Reddit. r/unixporn - the home for *NIX customization! https://www.reddit.com/r/unixporn/, 2023. Accessed: Feb. 27, 2023.

[117] Rémi Prévost, Mike McQuaid, and Danielle Lalonde. The Missing Package Manager for macOS (or Linux) — Homebrew. https://brew.sh/, 2022. Accessed: Jun. 22, 2022.

[118] P. C. Rigby and A. E. Hassan. What Can OSS Mailing Lists Tell Us? A Preliminary Psychometric Text Analysis of the Apache Developer Mailing List. In *Fourth International Workshop on Mining Software Repositories*, MSR '07, 2007.

[119] Christoffer Rosen and Emad Shihab. What are mobile developers asking about? a large scale study using stack overflow. In *Empirical Software Engineering*. Springer, 2016.

[120] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. In *Journal of computational and applied mathematics*. Elsevier, 1987.

[121] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E Hassan. Understanding reuse in the android market. In *Int. Conf. on Program Comprehension*. IEEE, 2012.

[122] Avigit K Saha, Ripon K Saha, and Kevin A Schneider. A discriminative model approach for suggesting tags automatically for stack overflow questions. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013.

[123] Hiroaki Sakoe. Dynamic-programming approach to continuous speech recognition. In *1971 Proc. the International Congress of Acoustics, Budapest*, 1971.

[124] Mohammed SAYAGH, Noureddine Kerzazi, Bram Adams, and Fabio Petrillo. Software configuration engineering in practice interviews, survey, and systematic literature review. *IEEE Transactions on Software Engineering*, 2020.

[125] Michael Schröder and Jürgen Cito. An empirical investigation of command-line customization. *Empirical Software Engineering*, 2022.

[126] Subhasree Sengupta and Caroline Haythornthwaite. Learning with comments: An analysis of comments and community on Stack Overflow. In *Proceedings of the 53rd Hawaii International Conference on System Sciences*, 2020.

[127] Rifat Ara Shams, Waqar Hussain, Gillian Oliver, Arif Nurwidyantoro, Harsha Perera, and Jon Whittle. Society-oriented applications development: Investigating users' values from bangladeshi agriculture mobile applications. In *Int. Conf. on Software Engineering: Software Engineering in Society*. IEEE, 2020.

[128] Sergei Shcherban, Peng Liang, Amjed Tahir, and Xueying Li. Automatic identification of code smell discussions on stack overflow: A preliminary investigation. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2020.

[129] E. Shihab, Zhen Ming Jiang, and A. E. Hassan. On the use of internet relay chat (irc) meetings by developers of the gnome gtk+ project. In *Proceedings of the 6th International Working Conference on Mining Software Repositories*, MSR '09, 2009.

[130] Emad Shihab, Zhen Ming Jiang, and Ahmed E Hassan. Studying the use of developer irc meetings in open source projects. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, ICSM '09, pages 147–156. IEEE, 2009.

[131] Martin Siegumfeldt. emacs +flyspell. https://lists.gnu.org/archive/html/help-gnu-emacs/2002-09/msg00619.html, 2002. Accessed: Feb. 27, 2023.

[132] Will Snipes, Vinay Augustine, Anil R Nair, and Emerson Murphy-Hill. Towards recognizing and rewarding efficient developer work patterns. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[133] Will Snipes, Anil R Nair, and Emerson Murphy-Hill. Experiences gamifying developer adoption of practices and tools. In *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014.

[134] Wei Song, Mengqi Han, and Jeff Huang. IMGDroid: Detecting Image Loading Defects in Android Applications. In *Int. Conf. on Software Engineering*. IEEE, 2021.

[135] Abhishek Soni and Sarah Nadi. Analyzing comment-induced updates on stack overflow. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories*, MSR '19, pages 220–224. IEEE, 2019.

[136] Sulayman Sowe, Ioannis Stamelos, and Lefteris Angelis. Identifying knowledge brokers that yield software engineering knowledge in OSS projects. *Information and Software Technology*, 2006.

[137] Charles Spearman. The proof and measurement of association between two things. *The American Journal of Psychology (AJP)*, 1961.

[138] Ivan Srba and Maria Bielikova. Why is stack overflow failing? preserving sustainability in community question answering. *IEEE Software*, 2016.

[139] Margaret-Anne Storey, Alexey Zagalsky, Fernando Figueira Filho, Leif Singer, and Daniel M German. How social and communication channels shape and challenge a participatory culture in software development. *IEEE Transactions on Software Engineering*, 2016.

[140] Ruoxi Sun, Wei Wang, Minhui Xue, Gareth Tyson, Seyit Camtepe, and Damith C Ranasinghe. An empirical assessment of global COVID-19 contact tracing applications. In *Int. Conf. on Software Engineering*. IEEE, 2021.

[141] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[142] tabnine. Code Faster with AI Code Completions | Tabnine. https://www.tabnine.com, 2021. Accesed: Nov. 04, 2021.

[143] Rukma Talwadker and Deepti Aggarwal. Popcon: Mining popular software configurations from community. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6. IEEE, 2019.

[144] Qiongjie Tian, Peng Zhang, and Baoxin Li. Towards predicting the best answers in community-based question-answering services. In *Proceedings of the Seventh International AAAI Conference on Weblogs and Social Media*, 2013.

[145] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. How do programmers ask and answer questions on the web? (NIER Track). In *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[146] Andrew Truelove, Eduardo Santana de Almeida, and Iftekhar Ahmed. We'll Fix It in Post: What Do Bug Fixes in Video Game Update Notes Tell Us? In *Int. Conf. on Software Engineering*. IEEE, 2021.

[147] Gias Uddin, Foutse Khomh, and Chanchal K Roy. Mining api usage scenarios from stack overflow. *Information and Software Technology*, 122:106277, 2020.

[148] Valve. Welcome to Steam. https://store.steampowered.com/, 2022. Accessed: Jun. 22 2022.

[149] Dirk Van Der Linden, Pauline Anthonysamy, Bashar Nuseibeh, Thein Than Tun, Marian Petre, Mark Levine, John Towse, and Awais Rashid. Schrödinger's security: Opening the box on app developers' security rationale. In *Int. Conf. on Software Engineering*. IEEE, 2020.

[150] Bogdan Vasilescu, Alexander Serebrenik, Prem Devanbu, and Vladimir Filkov. How Social Q&A Sites Are Changing Knowledge Sharing in Open Source Software Communities. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '14. Association for Computing Machinery, 2014.

[151] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. In *Empirical Software Engineering*. Springer, 2020.

[152] Diane Walker and Florence Myrick. Grounded theory: An exploration of process and procedure. In *Qualitative health research*. Sage, 2006.

[153] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond google play: A large-scale comparative study of chinese android app markets. In *Internet Measurement Conference 2018*, 2018.

[154] Haoyu Wang, Xupu Wang, and Yao Guo. Characterizing the global mobile app developers: a large-scale empirical study. In *Int. Conf. on Mobile Software Engineering and Systems*. IEEE, 2019.

[155] Peipei Wang, Chris Brown, Jamie A Jennings, and Kathryn T Stolee. Demystifying regular expression bugs. In *Empirical Software Engineering*. Springer, 2022.

[156] Shaowei Wang, Tse-Hsun Chen, and Ahmed E Hassan. How do users revise answers on technical Q&A websites? A case study on Stack Overflow. *IEEE Transactions on Software Engineering*, 2018.

[157] Shaowei Wang, Tse-Hsun Chen, and Ahmed E Hassan. Understanding the factors for fast answers in technical Q&A websites. *Empirical Software Engineering*, 2018.

[158] Shaowei Wang, David Lo, Bogdan Vasilescu, and Alexander Serebrenik. EnTagRec: An Enhanced Tag Recommendation System for Software Information Sites. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, 2014.

[159] Shaowei Wang, David Lo, Bogdan Vasilescu, and Alexander Serebrenik. EnTagRec++: An enhanced tag recommendation system for software information sites. *Empirical Software Engineering*, 2018.

[160] ArchLinux Wiki. Dotfiles. https://wiki.archlinux.org/title/Dotfiles, 2023. Accessed: Feb. 21, 2023.

[161] Wikimedia. Wikimedia users. https://strategy.wikimedia.org/wiki/Wikimedia_users, 2023. Accessed: Mar. 07, 2023.

[162] Wikipedia. Electronic AppWrapper - Wikipedia. https://en.wikipedia.org/wiki/Electronic_AppWrapper, 2022. Accessed: Jun. 22, 2022.

[163] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. In *Chemometrics and intelligent laboratory systems*. Elsevier, 1987.

[164] WordPress. WordPress Plugins | WordPress.org. https://wordpress.org/plugins/, 2022. Accessed: Jun. 22, 2022.

[165] Huayao Wu, Wenjun Deng, Xintao Niu, and Changhai Nie. Identifying key features from app user reviews. In *Int. Conf. on Software Engineering*. IEEE, 2021.

[166] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Tag recommendation in software information sites. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013.

[167] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '15. Association for Computing Machinery, 2015.

[168] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[169] Bo Yang, Zhenchang Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shanping Li. Don't do that! hunting down visual design smells in complex uis against design guidelines. In *Int. Conf. on Software Engineering*. IEEE, 2021.

[170] Di Yang, Aftab Hussain, and Cristina Videira Lopes. From query to usable code: an analysis of stack overflow code snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016.

[171] Shao Yang1 Yuehan Wang2 Yuan Yao and Haoyu Wang4 Yanfang Fanny Ye5 Xusheng Xiao. DescribeCtx: Context-Aware Description Synthesis for Sensitive Behaviors in Mobile Apps. In *Int. Conf. on Software Engineering*. IEEE, 2022.

[172] Deheng Ye, Zhenchang Xing, and Nachiket Kapre. The structure and dynamics of knowledge network in domain-specific q&a sites: a case study of stack overflow. *Empirical Software Engineering*, 2017.

[173] Jiaming Ye, Ke Chen, Xiaofei Xie, Lei Ma, Ruochen Huang, Yingfeng Chen, Yinxing Xue, and Jianjun Zhao. An empirical study of GUI widget detection for industrial mobile games. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2021.

[174] Shengcheng Yu, Chunrong Fang, Zhenfei Cao, Xu Wang, Tongyu Li, and Zhenyu Chen. Prioritize crowdsourced test reports via deep screenshot understanding. In *Int. Conf. on Software Engineering*. IEEE, 2021.

[175] Shengcheng Yu, Chunrong Fang, Yexiao Yun, and Yang Feng. Layout and image recognition driving cross-platform automated mobile testing. In *Int. Conf. on Software Engineering*. IEEE, 2021.

[176] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. Example overflow: Using social media for code recommendation. In *2012 Third International Workshop on Recommendation Systems for Software Engineering*, pages 38–42. IEEE, 2012.

[177] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *Int. Conf. on Software Engineering*. IEEE, 2021.

[178] H. Zhang, S. Wang, T. Chen, and A. E. Hassan. Reading Answers on Stack Overflow: Not Enough! *IEEE Transactions on Software Engineering*, 2019.

[179] H. Zhang, S. Wang, T. P. Chen, Y. Zou, and A. E. Hassan. An empirical study of obsolete answers on stack overflow. *IEEE Transactions on Software Engineering*, 2021.

[180] Haoxiang Zhang, Shaowei Wang, Tse-Hsun Chen, and Ahmed E Hassan. Are comments on Stack Overflow well organized for easy retrieval by developers? *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2021.

[181] Xueling Zhang, Xiaoyin Wang, Rocky Slavin, Travis Breaux, and Jianwei Niu. How does misconfiguration of analytic services compromise mobile privacy? In *Int. Conf. on Software Engineering*. IEEE, 2020.

[182] Zhen Zhang, Yu Feng, Michael D Ernst, Sebastian Porst, and Isil Dillig. Checking conformance of applications against GUI policies. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2021.

[183] Tianming Zhao, Chunyang Chen, Yuanning Liu, and Xiaodong Zhu. GUIGAN: Learning to Generate GUI Designs Using Generative Adversarial Networks. In *Int. Conf. on Software Engineering*. IEEE, 2021.

[184] Yuan Gao Zhen Wei and Jingqing Zhang. Automating question-and-answer session capture using neural networks. In *2019 KDD Workshop on Deep Learning for Education*, DL4Ed, 2019.

[185] Wei Zheng, Ricardo Bianchini, and Thu D Nguyen. Massconf: automatic configuration tuning by leveraging user community information. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*, 2011.

[186] Jiayuan Zhou, Shaowei Wang, Cor-Paul Bezemer, and Ahmed E. Hassan. Bounties on technical q&a sites: A case study of stack overflow bounties. *Empirical Software Engineering*, 06 2019.

[187] Shurui Zhou, Jafar Al-Kofahi, Tien N. Nguyen, Christian Kästner, and Sarah Nadi. Extracting configuration knowledge from build files with symbolic analysis. *Proceedings of the 3rd International Workshop on Release Engineering (RELENG '15)*, 2015.

APPENDICES

Appendix A

Literature Overview of Recent Research Involving App Stores

We reviewed relevant recent papers from the two flagship software engineering research conferences: the *ACM/IEEE International Conference on Software Engineering* ("ICSE") and the *ACM SIGSOFT International Symposium on the Foundations of Software Engineering* ("FSE"). We used *Google Scholar* to find papers containing the keyword "app store" between January 2020 and April 2022 for the two conferences. In Table A.1, we list the papers by how are app store involved and the corresponding app store if applicable.

Table A.1: Recent papers on app stores

Loc	Paper	Store
Mining software applications		
ICSE '21	Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications [177]	Google Play
ICSE '20	How does misconfiguration of analytic services compromise mobile privacy? [181]	Google Play
FSE '21	Algebraic-datatype taint tracking, with applications to understanding Android identifier leaks [112]	Google Play
FSE '20	Code recommendation for exception handling [99]	Google Play
FSE '20	Static asynchronous component misuse detection for Android applications [104]	F-Droid, Google Play, Wandoujia App Store
ICSE '21	Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection [11]	Google Play
ICSE '22	DescribeCtx: Context-Aware Description Synthesis for Sensitive Behaviors in Mobile Apps [171]	Google Play
ICSE '20	Time-travel testing of android apps [41]	Google Play
ICSE '20	An empirical assessment of security risks of global android banking apps [29]	Google Play, APKMonk, etc.
ICSE '21	Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code [6]	Google Play
ICSE '20	Accessibility issues in android apps: state of affairs, sentiments, and ways forward [7]	Google Play
ICSE '21	Don't do that! hunting down visual design smells in complex uis against design guidelines [169]	Android
ICSE '21	Identifying and characterizing silently-evolved methods in the android API [82]	Google Play
ICSE '21	Layout and image recognition driving cross-platform automated mobile testing [175]	Apple's App Store, Google Play
FSE '21	An empirical study of GUI widget detection for industrial mobile games [173]	Android Games
ICSE '21	Fine with "1234"? An Analysis of SMS One-Time Password Randomness in Android Apps [83]	Google Play, Tencent Myapp
ICSE '21	IMGDroid: Detecting Image Loading Defects in Android Applications [134]	Android
ICSE '21	GUIGAN: Learning to Generate GUI Designs Using Generative Adversarial Networks [183]	Android
ICSE '20	Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning [27]	Google Play
FSE '21	Frontmatter: mining Android user interfaces at scale [76]	Google Play
Mining app store non-technical attributes		
ICSE '20	Schrödinger's security: Opening the box on app developers' security rationale [149]	Apple's App Store, Google Play
ICSE '20	Scalable statistical root cause analysis on app telemetry [91]	Facebook App
ICSE '21	An empirical assessment of global COVID-19 contact tracing applications [140]	Android
ICSE '21	We'll Fix It in Post: What Do Bug Fixes in Video Game Update Notes Tell Us? [146]	Steam
ICSE '21	Automatically matching bug reports with related app reviews [59]	Google Play
ICSE '21	Prioritize crowdsourced test reports via deep screenshot understanding [174]	Android
ICSE '21	A first look at human values-violation in app reviews [101]	Google Play
ICSE '21	Does culture matter? impact of individualism and uncertainty avoidance on app reviews [48]	Apple's App Store
ICSE '21	COVID-19 vs social media apps: does privacy really matter? [60]	Google Play, Apple's App Store
ICSE '20	Society-oriented applications development: Investigating users' values from bangladeshi agriculture mobile applications [127]	Google Play
FSE '21	Checking conformance of applications against GUI policies [182]	Android
ICSE '21	Identifying key features from app user reviews [165]	Apple's App Store
ICSE '21	Champ: Characterizing undesired app behaviors from user comments based on market policies [66]	Google Play, Chinese android app stores
ICSE '20	Caspar: extracting and synthesizing user stories of problems from app reviews [58]	Apple's App Store