The Journal of Systems & Software 193 (2022) 111458

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Technical debts and faults in open-source quantum software systems: An empirical study ☆

Moses Openja *, Mohammad Mehdi Morovati, Le An, Foutse Khomh, Mouna Abidi

SWAT Lab, École Polytechnique de Montréal, Canada

A R T I C L E   I N F O

Article history:
Received 19 June 2021
Received in revised form 26 June 2022
Accepted 25 July 2022
Available online 30 July 2022

Keywords:
Quantum computing
Technical debts
Software bugs
Software maintenance
Software reliability

A B S T R A C T

Quantum computing is a rapidly growing field attracting the interest of both researchers and software developers. Supported by its numerous open-source tools, developers can now build, test, or run their quantum algorithms. Although the maintenance practices for traditional software systems have been extensively studied, the maintenance of quantum software is still a new field of study but a critical part to ensure the quality of a whole quantum computing system. In this work, we set out to investigate the distribution and evolution of technical debts in quantum software and their relationship with fault occurrences. Understanding these problems could guide future quantum development and provide maintenance recommendations for the key areas where quantum software developers and researchers should pay more attention. In this paper, we empirically studied 118 open-source quantum projects, which were selected from GitHub. The projects are categorized into 10 categories. We found that the studied quantum software suffers from the issues of code convention violation, error-handling, and code design. We also observed a statistically significant correlation between code design, redundant code or code convention, and the occurrences of faults in quantum software.

1. Introduction

Quantum Computing is a paradigm that intersects computer science, mathematics, and physics (Spector et al., 1999; Kaye et al., 2007). Unlike other computing fields, quantum computing uses the law of quantum mechanics with the goal to achieve high computation efficiency. In contrast, *quantum software* can be software applications that run quantum algorithms, provide a platform for testing and simulating quantum algorithms, or govern quantum computers' operations. The example of operational quantum software include applications developed for checking and correcting errors, maintaining the stability of quantum computers or software for supporting complex and highly computational tools such as medical equipment.

Software maintenance support is considered one of the most important aspects of the software development process. It is a well-known fact that between 60% and 80% of the total software development cost is taken by the maintenance phase (VanDoren, 1997). Previous studies have shown that modifying and revising previously released software's versions are the main activities of software maintenance (IEEE Standards Coordinating Committee, 1990; Krüger and Mauerer, 2020; Pérez-Castillo, 2020). Therefore, to ensure a maintainable software, developers have to design and implement the source code, in a way that facilitates future modifications and evolution changes. However, maintenance activities have never been a straightforward task, it is considered as the most time/effort-consuming and complex task throughout the life cycle of software development (Pigoski, 1996). The maintenance practices for traditional software systems have been extensively studied (Ko et al., 2006; Gyimothy et al., 2005; Xiong et al., 2009; Morales et al., 2017; Raymond, 1999; Pigoski, 1996; Khomh et al., 2009; Lenarduzzi et al., 2019; Li et al., 2015; Cataldo et al., 2008; Saika et al., 2016). These studies focus on a broad range of areas, including examining source-code complexity and software enhancement effort (Banker et al., 1998), estimating the reliability in the maintenance phase (Mateen and Akbar, 2016), studying the distributions and evolution of technical debts (Martin, 2019b; Martini et al., 2015a; Reimanis and Izurieta, 2016; Kruchten et al., 2012), and predicting the prevalence of bugs using anti-patterns (Taba et al., 2013; Ubayawardana and Karunaratna, 2018; Cairo et al., 2018) among others.

To the best of our knowledge, there is no prior study exploring maintenance efforts for quantum software yet. The recent emerging studies (Pérez-Castillo, 2020; Krüger and Mauerer, 2020) related to quantum software systems maintenance mostly focused on re-engineering new quantum algorithms within traditional software systems. For example, Pérez-Castillo (2020) proposed a model-driven re-engineering (Seacord et al., 2003) that allows

☆ Editor: Gabriele Bavota.

* Corresponding author.
E-mail addresses: openja.moses@polymtl.ca (M. Openja), mehdi.morovati@polymtl.ca (M.M. Morovati), le.an@polymtl.ca (L. An), foutse.khomh@polymtl.ca (F. Khomh), mouna.abidi@polymtl.ca (M. Abidi).

https://doi.org/10.1016/j.jss.2022.111458

the migration of classical or legacy systems together with quantum algorithms and the integration of new quantum software during the re-engineering of classical or legacy systems while preserving knowledge. While maintenance effort is a broader dimension, examining the maintenance in terms of the technical debt composition in a quantum software system is one direction to understand how this software is being maintained. Studies (Martini et al., 2015a; Cunningham, 1992a; Lenarduzzi et al., 2019; Li et al., 2015; Martin, 2019a) have shown that technical debts provide relevant and actionable insight into the design and implementation deficiencies of software systems. For instance, according to Martin (2019b), technical debts indicate the internal quality issues of software that make the software more difficult to modify or develop (for example the source code conventions for readability Smit et al., 2011). Understanding the distribution and evolution of technical debts in quantum software could guide future development and provide maintenance recommendations for the key areas that may require further attention to both the practitioners and researchers in quantum computing. In this paper, we examine the distribution and evolution of technical debts in quantum software and their relation with fault occurrences. In particular, we answered the following research questions:

(**RQ1**) **What Are the Characteristics of Technical Debts in Quantum Software?**
We examined the distribution of technical debts in quantum software systems represented as code smells and coding errors and their severity (categorize as critical, major, minor, and blocker) (Sonarqube, 2020). We summarized the technical debts based on the types of technical debts and highlighted the critical debts. Results show that about 80% of the technical debts are related to the code smells and more than half of technical debts in all software types belong to the major severity. The major severity are quality issues or flaw that can highly impact the productivity of developer, for example, an uncovered piece of code, unused parameters, or duplicated blocks. In addition, we found that a few types of technical debts (such as 'code convention' (problem with coding convention such as formatting, naming, white-space), 'design issues' (*e.g.,* duplicate string literals), 'brain-overload' (related to cognitive complexity), and 'error-handling') dominate the total number of technical debts.

(**RQ2**) **How Do Technical Debts Evolve Over Time?**
We investigated how new technical debts are added into the code-base with respect to the total file size over time. We observe that technical debts tend to be added in the initial versions of a project (when most new codes and files are added). Besides, we found that Lines of Code (LOC) can be considered as key indicators of the existence of technical debts in quantum computing software systems. This result is in line with the studies on traditional software (Molnar and Motogna, 2017; Molnar et al., 2019). We recommend quantum software developers pay more attention to the code quality and code size, especially when new files are added to the code base.

(**RQ3**) **What Is the Relationship Between Technical Debts and Faults?**
In this research question, we used regression models to examine the correlation between technical debts (and their types SonarQube, 2021b) and fault-inducing commits in quantum software at the file level. Our results indicate a statistically significant correlation. Particularly, we found that the highest significance in all studied quantum software systems is related to 'convention' and 'unused' technical debts.

The rest of this paper is organized as follows. In Section 2, we provide the background of this study. Then, we discuss the related works on quantum computing, technical debts, and fault analysis in the software systems in Section 3. We describe the methodology that is followed in Section 4. In Section 5, we present the results of our analysis. In Section 6, we discuss the implication of our findings. Section 7 describes the threats to the validity of our study. Finally, we provide the conclusion in Section 8.

2. Background

The main focus of this study is to investigate technical debts and their evolution in quantum software and how the technical debts affect the reliability of quantum software. This section describes the background of quantum computing and quantum software development.

2.1. Quantum computing

Quantum computers can adopt a superposition form of $|0\rangle$ and $|1\rangle$, which enables a qubit to exist in all of these states simultaneously. A qubit is a two-dimensional quantum-mechanical system, presenting sizeable information to process in quantum computing. This basic concept of superposition allows quantum computers to perform computations on an extensive scale in parallel (also known as parallel computation).

Another property of a quantum computer is the entanglement: given a two-qubit system, it is possible that the state of each qubit cannot be described separately. In mathematical form, the state cannot be factorized as the tensor product of two separate states. Certain types of highly entangled systems are challenging to model for classical computers. Generally, it is also possible to efficiently describe systems that do not exhibit too much entanglement using classical computational methods such as tensor networks (Orús, 2014). However, highly entangled systems are almost impossible to be modeled in classical computers. Consequently, quantum algorithms must exploit high amounts of entanglement to reach higher possible capabilities than classical algorithms. This entanglement property has applications in many aspects of quantum computing, such as cryptography (Ekert, 1991), and quantum computation (Nielsen and Chuang, 2002; Shor, 1999).

2.2. Quantum software development

Zhao (2020) defined Quantum software engineering as *"the application of sound engineering principles for the development, operation, and maintenance of quantum software and the associated document to achieve economically quantum software that operates efficiently on quantum computers and is reliable"*.

Like classical software, quantum software is also developed by following a series of steps: requirements analysis, design, and implementation to testing and maintenance. The quantum design stage provides the means for modeling and defining quantum software systems (Kiefl and Hagel, 2020; Pérez-Delgado and Perez-Gonzalez, 2020). The testing phase aims to identify any flaws or defects in the quantum software and verify that the software's behavior reflects the documentation defined at the early phase of analysis. Finally, the maintenance as the last phase of quantum development process represents any changes or updates later when quantum software products are released. Indeed, to develop maintainable and reliable quantum software systems, there are numerous factors that developers must take into consideration and respect in every phase of software development, for example, defining system requirements and quantum design patterns, implementing test and fault detection models, implementing development environments, and selecting quantum programming mechanics. For further readings on quantum software development, we refer our readers to Zhao (2020), Pérez-Delgado and Perez-Gonzalez (2020), Thompson et al. (2018), Nielsen and Chuang (2002) and Piattini et al. (2020b) containing well documented set of guidelines to assist a well-engineered quantum software system development.

Quantum programming is the design and implementation of a program executable on a quantum computer to meet the computing need (Ying, 2016; Miszczak, 2012). Every chunk of code in a typical quantum program consists of both quantum and classical instructions (Cook and Mitchell, 1997). A classical instruction uses classical bit registers to measure the qubit states, including conditional operation, whereas quantum instruction uses qubit registers to operate on the quantum computer. Unlike in the early stage of Quantum Turing Machine (QTM) (Deutsch, 1985), quantum programming majorly focuses on quantum circuit models. This is followed by new quantum programming models such as the Quantum Random-Access Machine (QRAM) model (Knill and Nielsen, 2000) and pseudocode (Knill, 1996). Instead of just designing the quantum circuit, a quantum program is built to run on a classical computer to control the quantum system. Numerous other quantum programming languages such as 'Quipper' (Green et al., 2013), Q# (Svore et al., 2018) and 'Scaffold' (JavadiAbhari et al., 2015; Abhari et al., 2012) are now available. These programming languages are built on top of the traditional programming languages such as C#, Python, Java, C/C++, and Julia. For the complete list of the programming languages and their history, we refer the readers to the studies (Cook and Mitchell, 1997; Zhao, 2020; Ying, 2016).

3. Related works

In this section, we discusses the related works on quantum computing, technical debts, and analysis of fault characteristics.

3.1. Quantum computing

Piattini et al. (2020a) studied the emergence of quantum computing software systems and quantum software engineering. They also explained that based on evidences, demand for quantum software systems will be increased dramatically during the next years of the current decade. Besides, they mentioned nine principles as the main principles of quantum software engineering.

Garhwal et al. (2019) studied various high-level quantum programming languages and identified the main features of each one. They categorized quantum programming languages into five different main classes (such as Quantum Object Oriented Programming Language, Quantum Circuit Language, etc.). Next, they represented that QPL and QFC are the most popular quantum programming languages.

Moguel et al. (2020) explained that quantum software systems in quantum software engineering need processes that require methodologies, like classical software systems. They identified that classical software engineering processes such as requirements specification, architectural design, detailed design, implementation or testing can be used to carry out activities of each process in quantum software engineering effectively. While, the methodologies for each process of quantum software engineering should be adopted based on the requirements of quantum software development. The reason behind this difference is the underlying model of computing being used in each one. In classical software systems, computation is done by a sequence of instructions manipulating the data and the final state should be the output of the program. Although in quantum computing there is not a sequence of instruction. System in quantum computing has a set of states and can be in all of them at the same time. Besides, system stops when a certain subset of system states is in the desired state.

The most related work is a recent study regarding engineering quantum software (Zhao, 2020). In this paper, the authors introduced a quantum software life cycle and named it as *quantum software engineering*. They firstly explained quantum programming as the process of designing and building an executable quantum computer program. Quantum Software Requirements Analysis, Quantum Software Design, Quantum Software Implementation, Quantum Software Testing, and Quantum Software Maintenance have been reported as the main stages of engineering quantum software systems. It is also mentioned that quantum software testing is more difficult than classical counterparts that are resulted from the structure of quantum computing programs and also quantum computers. Seven different types of faults (Incorrect quantum initial values, Incorrect operations and transformations, Incorrect composition of operations using iteration, Incorrect composition of operations using recursion, Incorrect composition of operations using mirroring, Incorrect classical input parameters, and Incorrect deallocation of qubits) are introduced as faults related to quantum software engineering to achieve a deep understanding about the behavior of faults in quantum computer programs. To detect introduced faults, an assertion has been explained for each identified fault category.

Another related work to our study is the article discussing open-source software in quantum computing carried out by Fingerhuth et al. (2018). In that article, a wide range of open-source software systems focusing on quantum computing are studied and the authors introduced four paradigms to develop quantum computing software projects, in order to ease the understanding of quantum computing systems for computer scientists and software engineers. The first paradigm is the discrete variable gate-model quantum computing paradigm in which bits and logical transformations are replaced by qubits and a finite set of unitary gates respectively. Continuous variable gate-model quantum computing is another paradigm where the qubits are replaced by qumods taking continuous values. This paradigm is mostly regarding physics aspects of quantum mechanics and particularly, quantum optics. The third introduced paradigm is Adiabatic quantum computation that uses adiabatic theorem, a phenomenon from quantum physics, to find the global optimum of a discrete optimization problem. Last but not least, they discuss quantum simulators which are application-specific quantum devices. But this paradigm is different from the simulation of quantum computations. The author of this paper just reviewed the development process of quantum computing projects from a software engineering view point.

Shaydulin et al. (2020) focused on the contributors of open-source quantum computing projects hosted on GitHub. They studied data of 146 contributors and surveyed 46 of them. Based on the analysis of the collected data, they mentioned that almost all (45 out of 46) of the contributors of quantum computing software systems did not receive formal training in quantum computation. Thus, they concluded that their lack of a good understanding of quantum computing may lead to poor quality software systems. Besides, they introduced the main challenges that developers are facing in open-source quantum computing software systems.

3.2. Technical debts

Cunningham (1992b) introduced the concept of technical debt (TD) for the first time in 1992. Since that time, a number of studies have been carried out to shed light on the different properties

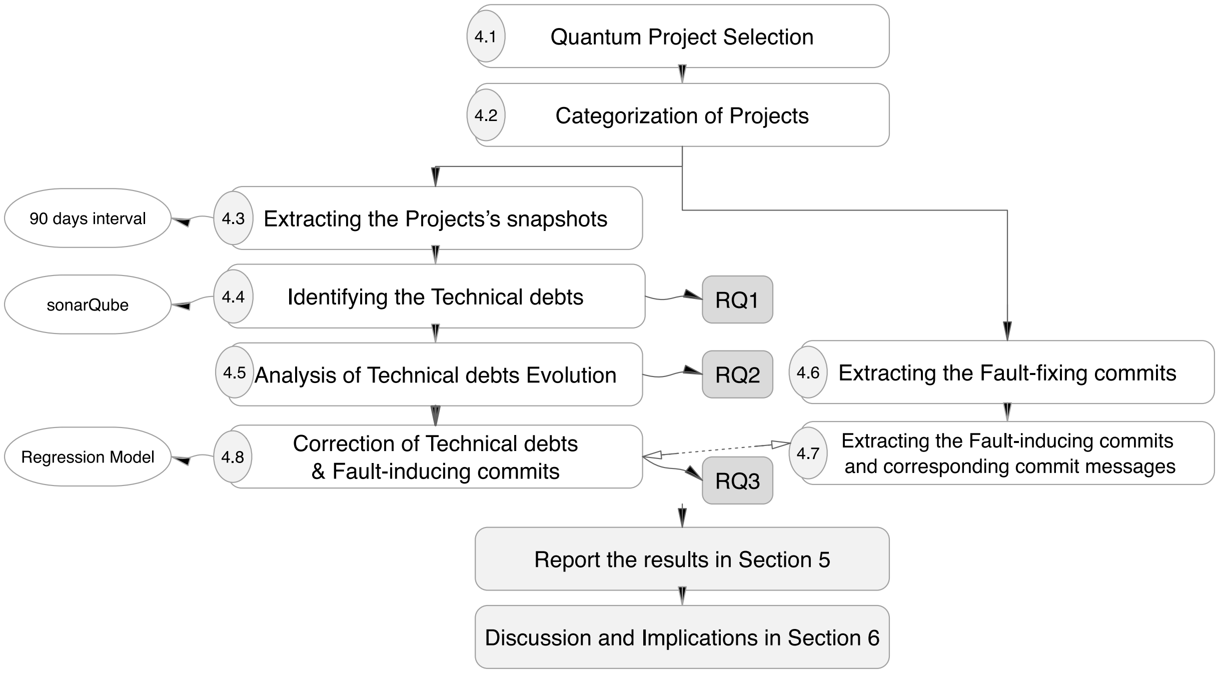


Fig. 1. An overview of our study methodology.

of TD. Avgeriou et al. (2016) found that although TD can yield some benefit in short term, it can increase the cost of changes in a long term. Martini et al. (2015b) showed that the existence of TD in software systems is inevitable. Besker et al. (2019) studied the strategies to prioritize the identified TDs to be resolved and the most important factors affecting this process. Besker et al. (2018) carried out another study to determine the refactoring cost and negative effects of different types of TD. Besides, there have been various tools for identifying TDs. As an example, Avgeriou et al. (2020) conducted a comparison of 9 TD tools and highlighted the main features of each one. They showed that SonarQube can best handle multiple programming languages, which is the reason why we used this tool in our study because our subject Quantum software is written in different programming languages.

Technical debts in software imply quality issues, *i.e.,* code quality was ignored to achieve a faster goal instead of using a systematic approach to reach the same purpose that would take longer time (Techopedia, 2017). Therefore, this may involve additional rework such as code refactoring in a later time to achieve a better quality. For this reason, tools such as 'Squore1', 'SonarQube', 'Kiuwan2' or 'Ndepend3' have been proposed both in academia and industry to assist in identifying technical debts. Such tools are built based on a set of quality models (Samadhiya et al., 2010) to help practitioners reduce the time it takes to synthesize and make the best decision regarding a system's future quality by providing systematic and quantifiable metrics for the industry's best practice (Reimanis and Izurieta, 2016). For example, SonarQube, NDepend and Squore use the SQALE model (Letouzey, 2012) that is programming language independent, whereas tools such as Kiuwan uses the concept of Checking Quality Model (CQM).

4. Study design

This section presents the methodology we followed in this study. We used sequential mixed-methods to answer our research questions **RQ1** through **RQ3**, from data collection, data processing, to quantitative and qualitative analyses (Ivankova et al., 2006). Fig. 1 shows an overview of our methodology.

To avoid any ambiguity, we use the following terms throughout the paper to denote different kinds of defects identified by SonarQube and GitHub project's repositories.

- *Error*: Potential coding errors or bugs detected by SonarQube. These errors might break the code and have to be fixed as soon as possible.
- *Code smell*: They include poor coding decisions and any characteristics in the software source code that indicate the possibility of a deeper problem, as reported by SonarQube (Brown et al., 1998; Fowler and Beck, 1999).
- *Fault*: Post-release bugs or developer/QA reported bugs. Only the GitHub issues (GitHub, 2021) labeled as 'bug' will be considered as faults.

4.1. Selection of quantum projects

The first step of our work is to select a list of open-source quantum computing projects. We searched quantum computing projects from GitHub, because GitHub hosts the largest collection of open-source software. Some of the famous quantum projects, such as `Qiskit` from IBM, are shared on GitHub. In the following sections, we will use this pattern <author_name>/<repo_name> to denote a GitHub project, whose URL will be https://github.com/<author_name>/<repo_name>.

Searching against the Rest API (GitHub, 2021) provided by GitHub, we obtained a total of 1364 repositories that: (1) contain the word 'quantum' (case insensitive in either the repository name, descriptions, or project ReadME file); (2) are a mainline, not a forked repository; (3) have a ReadME or description written in English to provide us details about the project. We used the GitHub API (`:owner/:repo`) to extract the repositories descriptions and other meta-data then used to manually check if the descriptions are written in English. We then use the following criteria to filter out our search results.

- Following the idea of previous works (Munaiah et al., 2017; Businge et al., 2018, 2019, 2022), we selected the repositories that have been forked more than once to reduce the chance of selecting a student's class assignment. This step removed 803 repositories and remained with 561 repositories.

- Our projects should contain enough history and development activities for the analysis of technical debts and faults. Thus, we only considered the repositories that have been created more than 10 months earlier (*i.e.,* 2019-09-16) than the date of this study. Inspired by Fingerhuth et al. (2018) and Openja et al. (2022), we chose the projects that have at least 100 commits and 10 GitHub issues or pull requests. In addition, we limited our selection of projects to those with at least one release. These selection criteria allowed us to consider projects targeting the end-users because 'never released projects' may not show representative data on faults experienced during the project development (Openja et al., 2020) or by the end-users. This step further removed 417 projects and remained with 144 projects.
- We manually read the project descriptions and removed 13 repositories that are only related to quantum documentations or lecture notes. We also identified and removed 9 more repositories that are not related to quantum computing but merely contain the word 'quantum' in their descriptions, such as `foxyproxy/firefox-extension` and `quantacms/quanta`. After this step, we finally obtained a list of 122 repositories.

4.2. Categorization of projects

In this step, we classified the selected quantum projects into different categories. We used the list of categories provided by the Quantum Open-Source Foundation (`QoSF`) (QoSF, 2021), which is an initiative to promote the advancement of open-source tools for quantum computing. To decide which category a project belongs to, we first checked whether a target project is listed by QoSF. If it is listed, we will directly use the QoSF provided category. Otherwise, two of the authors will independently read the project descriptions and/or the project's official website to classify the project into one of the QoSF defined categories. The two authors will compare their results and resolve all discrepancies until reaching an agreement for all of the projects.

During the categorization step, the authors removed four more projects either because their programming languages are not supported by SonarQube or because they are identified as experimental or toy projects, such as `Quantum-Game/quantum-game-2` and `PJavaFXpert/quantum-toy-piano-ibmq`. The remaining 118 projects are classified into 10 categories as follows. Readers can refer to Appendix A for a list of example repositories that belong to each of the categories.

- *Full-Stack Library or Framework:* This can be seen as all-in-one software containing all the frameworks and/or libraries required for building quantum applications.
- *Experimentation:* Tools that support experimentation of quantum systems or states, such as superconducting qubit systems and parametrization of a pulse.
- *Simulator:* Controllable quantum systems that enable users to study the quantum systems which are hard to study on actual hardware or in laboratory.
- *Cryptography:* This class is related to the usage of quantum mechanics' characteristics to carry out cryptography tasks.
- *Quantum-Algorithms:* Systems that execute a quantum computation on a quantum model (such as a quantum circuit model). They are designed majorly to solve the classical problems in a probabilistic fashion (Montanaro, 2016).
- *Toolkit:* A set of libraries and tools that help interact with different components of quantum systems mainly in research settings.
- *Quantum Annealing:* Systems that provide meta-heuristics for finding global minimum over a very large number of possible solutions by using quantum fluctuation-based computation (Finnila et al., 1994).
- *Quantum-Chemistry:* Projects that focus on the use of quantum mechanics in the chemical systems experiments.
- *Compiler:* When a quantum computing algorithm is implemented on actual hardware, the circuits should be compiled for the restricted topology of the particular quantum chip used for execution. This category is related to the systems translating quantum circuits to the quantum assembly format.
- *Assembly:* This class is related to quantum assembly languages used for describing quantum circuits. It is used in many quantum compilation and simulation tools as the intermediate representation to describe quantum circuits (Cross et al., 2017).

The summary statistics of the selected repositories in the 10 categories are shown in Fig. 2. The number of commits ranges from 101 to 9592 with the overall mean of 1101 and median of 532 commits.

4.3. Extraction of snapshots of studied projects

Our study aims to analyze the change history of the selected quantum projects to investigate the maintenance effort over time. *Git* allows us to take snapshots of a given project at a specific time period. We defined *snapshots* as different copies of the same project. Each snapshot contains a set of changes made at that point of the project development. We report the distribution of the commits in Fig. 2 for a given studied quantum category. We use the following steps to extract the snapshots.

Choosing the Snapshot Period. Studying snapshots from each of the commits will provide us with the most precise result, but this will also exhaust our computational resources, making it impossible to analyze all the 118 projects. Based on the idea of iterative and incremental development, we assume that developers of the studied projects did not make a large number of changes between consecutive commits. Thus, we can use a snapshot every N days to represent the changes and development activities (such as fault fixes) during these days. To decide the best N, we investigated the distribution of commits between days interval {30, 60, 120, 150, 180} to be able to choose the appropriate number of days and extract project snapshots with considerable code-changes across all the studied projects. Fig. 3(a) details how commits are distributed within the time frames of {30, 60, 120, 150, 180} days in the studied quantum categories.

For example, the box-plot corresponding to 30 indicates the number of commits every 30 days across all projects in the categories. Observing the commits distribution with the time-frames shown in the figure, we chose 90 days as the appropriate time frame, which represents the median size number of commits for most of the selected projects. This size of time frame also allows us to extract a number of snapshots from each project, which is feasible to conduct the following analyses based on our computational resources.

Extract Snapshots. Using the time frame of 90 days, we identified the latest commits for every time frame (snapshots). We then used git archive (i.e., https://github.com/<author_name>/<repo_name>/archive/<commit>.zip) to create and download a zip file containing only the files under git data source from the starting of the project until the latest commit in a given snapshot. Fig. 2 shows the distribution of number of snapshots (and other metrics such as age) extracted from the projects categories. Considering that most of the projects we analyzed have at least 212

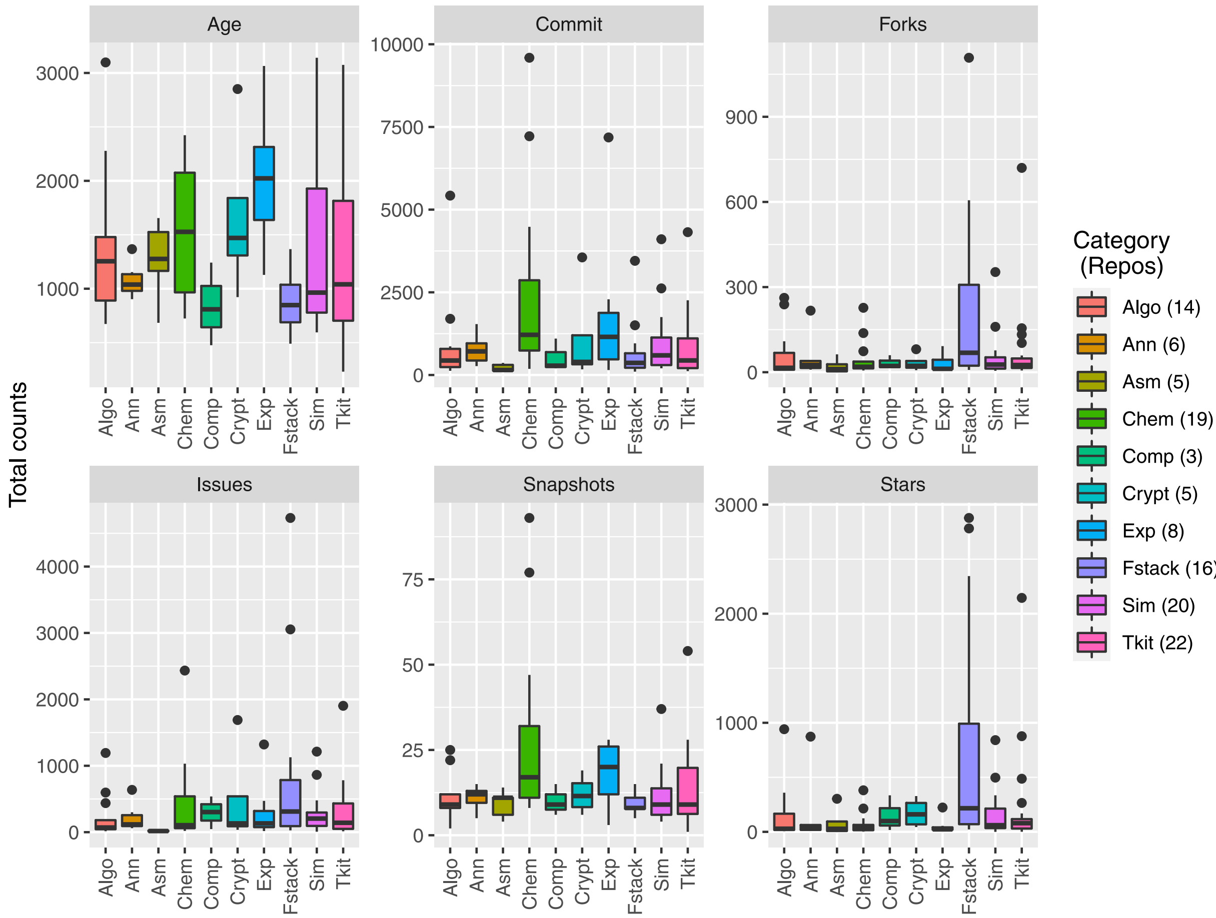


Fig. 2. Statistics of selected quantum projects by categories (*Commits*: number of commits, *Age*: Time (in days) from the project creation date to its latest change, *Issues*: number of issues reported on the GitHub repositories, *Forks*:number of forks (popularity metric), *Stars*:number of stars (popularity metric), *Snaps*:number of snapshots extracted based on 90 days (extracted in Section 4.3), *Repos*: number of studied GitHub repositories in respective quantum category.) **Algo**:Algorithms, **Ann**:Annealing, **Asm**:Assembly, **Chem**:Chemistry, *Comp*:Compiler, **Crypt**:Cryptography, **Exp**:Experimentation, **Fstack**:Full-stack library, **Sim**:Simulator, *Tkit*:ToolKit.

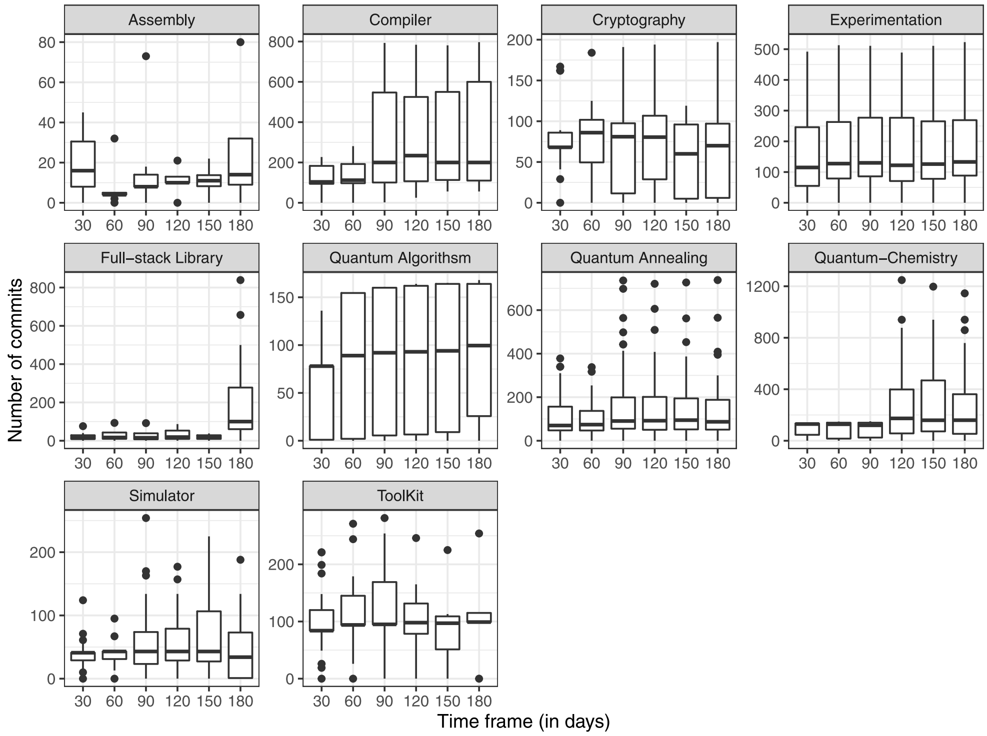
Table 1
The median of mean number of changed lines of code for every 10 consecutive snapshots for the projects in each quantum category.

Category	First 10 snapshots	Between 11 to 20 snapshots	Between 21 to 30 snapshots
Full-stack library	17,632	212	139
Compiler	198,001	127	6
Cryptography	20,652	332	7
Quantum-chemistry	143,297	47,974	102,077
ToolKit	42,027	4,389	1,666
Quantum algorithms	50,450	1,476	38
Experimentation	18,402	7,295	1,223
Simulator	57,982	12,291	507
Quantum annealing	3,764	91	22
Assembly	6,622	546	65
MEDIAN	**31,340**	**1,011**	**102**

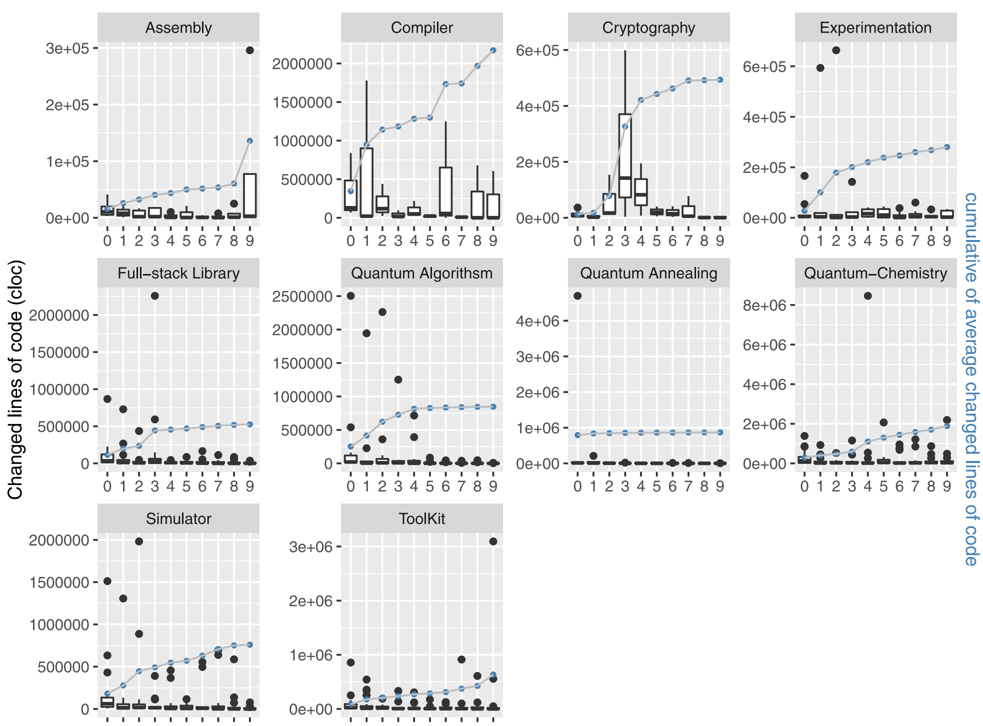
commits (with maximum commits of 9592) and the survival times (age) of 476 days at the point of starting this study. There is a trade-off when analyzing the number of revisions of each project of manageable size and code changes. Fig. 3(b) illustrates consecutive changes in terms of changed lines of code within the commits for the snapshots based on a 90 days interval (i.e., code changes for the selected time frame). For the clear visualization, we only show the consecutive changes for the first ten snapshots of each of the quantum category. In Fig. 3(b) we also show the cumulative mean value of the changed lines of code (right) across the snapshot for the selected time frame. Overall, according to Fig. 3(b), more changes are associated with the initial snapshots tend compared to the latest snapshots. Table 1, further details on the consecutive changes for the first 30 snapshots, sub-divide into three categories, i.e., first 10 snapshots, between 11 to 20 snapshots, and between 21 to 30 snapshots. As can be seen in Table 1 indicating that most development activity occur within the first 10 snapshots of the project and only fewer code changes made for the snapshots between 11 to 30 in 90% of the studied quantum categories.

4.4. Technical debts in quantum software

This section presents the steps carried out to identify technical debts from the snapshots extracted from the previous section. First, we give a general background of the SonarQube tool that we used to detect technical debts. Then we describe our steps for detecting the technical debts.



(a) The distribution of commits across different time frames (in days) for the studied categories of quantum projects. The y-axis indicates the number of commits, and the x-axis indicates the time frames (in days) from 30 to 180 days with the interval of 30 days.



(b) The distributions of lines of code for the first ten snapshot's of the studied projects, basing on the 90 days interval.

Fig. 3. The figures highlighting the distributions of commits and the consecutive changes. In the first part of the figures, we shows how the commits are distributions across different time frames from 30 to 180 (in days) for the studied categories of quantum projects. The second figure indicate the changes in terms of changed lines of code (left) and the cumulative mean value of the changed lines of code (right) across the first ten snapshot's consecutive commits for 90 days interval.

Table 2
Severity rules define by SonarQube (2021b).

Severity	Impact	Likelihood	Description
BLOCKER	✓	✓	Higher impact probability to the behavior of the application, for example memory leak or unenclosed JDBC connection. The code needs to be fixed immediately.
CRITICAL	✓	×	Either an error with a low probability to impact the behavior of the application or a security flaw such as an empty catch block or SQL injection.
MAJOR	×	✓	Quality issues or flaw that can highly impact the productivity of developer, for example, an uncovered piece of code, unused parameters, or duplicated blocks.
MINOR	×	×	Quality issues or flaw which can slightly affect the productivity of developers, for example, too long lines, or 'switch' statements with fewer than 3 cases.

4.4.1. SonarQube platform

We chose SonarQube in this study because: (1) It is broadly used for technical debts detection by over 200 thousand users (SonaQube, 2021) and academic research setting (Tan et al., 2020; Marcilio et al., 2019; Saarimäki et al., 2019; Digkas et al., 2017, 2018); (2) The tool is based on a SQUARE quality model (Letouzey, 2012; Letouzey and Coq, 2010), which is academically evaluated and published (Dale and Izurieta, 2014; Letouzey and Ilkiewicz, 2012); (3) Our subject projects are written in different programming languages, which are supported by SonarQube.

SonarQube is an open-source platform used for monitoring source code quality and security. It uses static analysis to detect code smells, potential bugs (referred to as *errors* in the rest of this paper), and vulnerability in software systems written with over 20 programming languages, including Python, C#, C/C++, JavaScript, XML, and Java. It also allows creating plugins to support new programming languages. SonarQube uses a rule definition during source code analysis. An alert is raised in case a rule is broken. The properties of the created alerts include the type, severity, and effort needed to fix the alert.

SonarQube uses a risk estimation procedure to assign severity levels to the detected debt. Table 2 illustrates the severity definition, extracted from the official SonarQube documentation (SonarQube, 2021b). The columns 'Impact' and 'Likelihood' indicate how SonarQube assesses whether the severity of the debt is high or low. For example, evaluating if the code error can cause the application to crash or corrupt stored data (Impact) and the probability that the worst (Bloch, 2003) will happen (likelihood), or assessing whether the detected code smell could lead to a error during the maintenance process. Also, SonarQube uses tags to categorize rules and debts. One or more tags are assigned to the debts inherited from the rules that raised them. Moreover, tags denote different types of technical debts. Also, each rule provides an estimation function for determining the time needed to fix the corresponding debts. The estimation usually offer either a function with a linear offset or a constant time per debt. For example, the rule 'S3776' states that 'the Cognitive Complexity should not be too high'. It generates code smells of the severity type *CRITICAL* with a linear time to fix, which consists of a constant 5 min time per issue, including one more minute for each additional complexity point over an established threshold.

Also, as part of the analysis results returned from running SonarQube, SonarQube calculates the effort required (in terms of time) for fixing technical debts (TD) in a target system. Further, SonarQube computes the technical debts ratio as $TDR = \frac{Td}{devT}$; where $devT$ is the time estimated to develop the system, where a single line of code (LOC) is estimated to take 30 min. Then the *TDR* is classified on a scaling of A as the best ($TDR < 5\%$) to E as the worst ($TDR \geq 50\%$). This detailed information, therefore, gives a high-level view of the system's internal quality.

The characteristics of technical debts important in our study are: (1) The types of debts: code smells (maintainability domain) and coding error (reliability domain). (2) The severity of technical debts, introduced in Table 2. (3) The effort required to fix the technical debts mentioned above. Providing these breakdowns can assist practitioners in prioritizing the allocation of resources to address critical debts.

4.4.2. Detection and analysis of technical debts

We ran SonarQube on every project's snapshots extracted in Section 4.3. The tool is configured on a local computer and used its web interface to monitor the results of the analysis. For data extraction and further analysis of the SonarQube results, we used a purpose-written program written with Python and Java (i.e., code written specifically to achieve the analysis goals guided by our RQ), making use of the SonarQube API and spreadsheet software as the primary storage. Fig. 4 highlights the general summary of the metrics obtained after running SonarQube analysis.

To answer our **RQ1**, we investigated technical debts for every snapshot of the target project. We used the Spearman test (Zar, 2005) with a high correlation value (>90) between thousands of lines of code (KLOC) which was used as the proxy for application size and the total classes and functions for all 118 target projects. In Section 5.1, we will discuss the results of this analysis.

4.5. Evolution of technical debts

Previous studies (Martini et al., 2015b; Molnar and Motogna, 2017) have shown that refactoring efforts and architectural changes are some of the factors that influence technical debt in software projects compared to source code file size. Also, constraints such as time and budget have been described as some of the root causes of technical debt (Lenarduzzi et al., 2019). This step examines how technical debts (in terms of errors and code smells) evolve across the snapshots of every target project to help us understand how debt is introduced in quantum software source code. Specifically, we computed the technical debt ratio (*TDR*, described in Section 4.4.1) for the debts detected in each snapshot of the target projects. We also examined the addition of new codes (in KLOC) and how the codebase grows to identify the point in time when debts are introduced. Finally, we verify how the source code file size impacts the technical debts using a Spearman rank correlation. We present the detailed results of this analysis in Section 5.2 to answer our **RQ2**.

4.6. Identification of fault-fixing commits

One of the goals of this study is to examine activities that may introduce faults during quantum software development. To achieve this goal, we analyzed the fault-fixing and fault-inducing commits and their correlation with the overall technical debts discussed in the previous section. We defined a *fault-fix commit* as code changes to fix faults and a *fault-inducing commit* as the code-changes that induced faults (Wen et al., 2019).

To identify the fault-fixing commits, we combine two approaches: (1) using a list of keywords such as "bug" and "fixes" as shown in Listing 1. For this case, we checked the presence

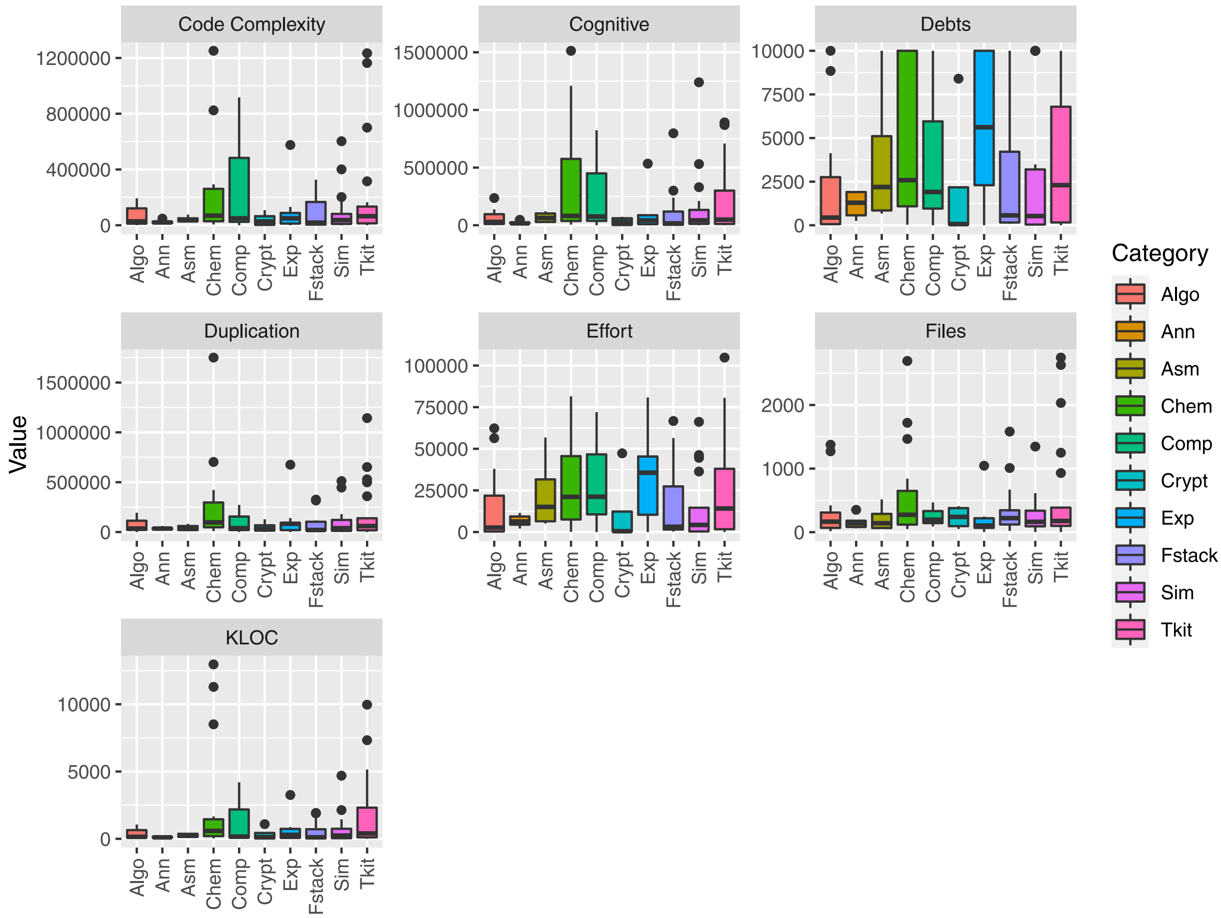


Fig. 4. Summary of the analysis results from SonarQube (*Debts*: number of technical debts detected, *Effort*: estimated time (in minutes) to fix all reported technical debts, *Complexity*: reported source-code cognitive complexity, *KLOC*: number of thousand lines of code for the analyzed files, *Files*: number of unique files with at least one technical debt) **Algo**:Algorithms, **Ann**:Annealing, **Asm**:Assembly, **Chem**:Chemistry, *Comp*:Compiler, **Crypt**:Cryptography, **Exp**:Experimentation, **Fstack**:Full-stack library, **Sim**:Simulator, **Tkit**:ToolKit.

of at least one of the keywords in Listing 1. Our selection of keywords is based on previous studies (Mockus and Votta, 2000; Zhong and Su, 2015; Kim et al., 2007; Abidi et al., 2021) and (ii) using the identifier/references to the bug reports (BR) within the commit message (i.e., references to the issue labeled as 'bug') such as #131 (Čubranić and Murphy, 2003; Fischer et al., 2003; Śliwerski et al., 2005; Zhong and Su, 2015; Kim et al., 2007). The two approaches above consider the faults that are reported in the bug tracking system (for our case GitHub issues tracker) and those that are not reported in the system, because developers might fix a fault in the source code which is not captured by such systems (also known as 'on-demand') (Zhong and Su, 2015).

Using the two approaches described above we obtained a total of 33,917 unique fault-fixing commits that modified 49,593 unique files across all the studied projects. Next, we removed all the commits detected as fault-inducing and fault-fixing that touched only the files related to readMe/ documentation, git, or test cases. To this end, we checked if a given keyword (case insensitive) is within the file path or file name. In Listing 2 we show the list of keywords used to match if the modified files by the fault-fixing commits are related to documentation, licence, test case, or git-related files. Specifically we follow the idea from the previous studies (Herbold et al., 2022; Zhong and Su, 2015) to select the keyworks (such as test, documentation) for filtering out the fault positive results. After cleaning the commits related to the documentation or test cases, the final list of the fault-fixing commits is 30,044 unique fault-fixing commits with 39,285 unique files modified.

Listing 1: The keywords used to extract fault-fix commits

```
"fixed ", "fixes ", " fixed", "fix", "resolves", "crash",
    "fall back", "coverity", "reproducible",
    "stack-wanted", "failur", "fail", "npe ", "except",
    "broken", " bug", "error", "addresssanitizer", "hang
    ", "permaorange", "random orange", "intermittent",
    "steps to reproduce", "crash", "assertion",
    "failure", "leak", "stack trace", "heap overflow",
    "freez", " problem", " overflow", " issue",
    "workaround ", "break ", "stop"
```

Listing 2: The keywords used to removed the fault-fixing and fault-inducing commits related to documentation or test case

```
"readme", "doc", "test", ".png", ".pdf", ".jpeg", ".jpg",
    ".gif", "licence",
".txt", ".md", ".git", "changelog", ".rst", ".bib", ".json",
    "tox."
```

4.7. Extracting the fault-inducing commits

This step aims to identify the fault-inducing commits and use the information to check for the correlation with the debts in the quantum software systems. We may not directly know how faults are induced, but we can extract characteristics of the fault-inducing commits. We used the SZZ algorithm (Kim et al., 2006) to identify fault-inducing commits using the fault-fixing commits.

Given the fault-fixing commit, SZZ will track the fault-inducing predecessor lines to lines modified in the fixing commit within a software repository. As a step in SZZ, for every fault-fixing commit, SZZ will identify all previous commits that changed the same lines of code using `git blame` command, resulting in a set of a fault-inducing commits that might have introduced the fault. Next, the identified commit time is compared to the time when the corresponding bug reports were submitted to determine if it should be ruled out as fault-introducing or not. The commit created later than the submission time is inducing commit if it is either a partial fix- did not fully resolve the fault as evident in the later fault-fixing commit for the same fault or is responsible for another fault. Hence, separate fault-fixing commits may originate from similar fault-inducing commits that have modified related files. Many previous studies such as Bernardi et al. (2012), Asaduzzaman et al. (2012), Bavota et al. (2012), Canfora et al. (2011), Ell (2013), Eyolfson et al. (2011), Kamei et al. (2013), Kim and Whitehead (2006), Kim et al. (2008), Rahman and Devanbu (2011), undefinedliwerski et al. (2005), Tufano et al. (2017), Wen et al. (2016), Wu et al. (2018) and Yin et al. (2011) have also leveraged the SZZ algorithm to detect fault-inducing commits from a fault-fixing commit. Also, many tools have been proposed implementing the SZZ algorithm such as Pydriller (Spadini et al., 2018), Commit Guru (Rosen et al., 2015) or SZZ Unleashed (Borg et al., 2019). In this study, we used the Pydriller framework to detect the candidate fault-inducing commits because it is very convenient for mining software repositories and provides a set of APIs to extract important historical information regarding commits (Spadini et al., 2018). Where necessary, we treated the candidate fault-inducing identified using the fault-fix with issues reference (BR) differently from the fault-inducing commits that do not have issue reference (identified by 'on-demand' fault-fixing commit). For the BR, all the fault-inducing commits with date greater than issue reported date and do not satisfy the partial fix or are not responsible for another fault were considered a false positive and were removed. Similarly, we removed the fault-inducing commits with a date greater than the fault-fixing date for the on-demand fault-fixing commits. In total, we removed 234 on-demand commits and 731 BR commits and their respective fault-inducing.

To verify the accuracy of the extracted fault-inducing commits, we performed a manual analysis on the sampled fault-inducing commits. We randomly selected 382 fault-inducing commits based on 95% confidence interval and manually checked if the changes in the sampled fault-inducing commits are indeed related to the modifications performed in the corresponding fault-fixing commits. Two authors discussed and assigned each commits with tags as either "True", "False", or "Unclear". In this case, the tag "True" was assigned for a situation where the authors were convinced that the change performed in fault-fixing was indeed related to the changes applied in fault-inducing. While "False" was used for the case where the changes are not related. Finally, a tag "Unclear" was attached in situations where the authors could not completely relate to a tag "True" or "False". The two authors discussed any conflicts until reaching an agreement. We calculated the precision considering only the True and False tags for data emerging from our manual validation and found a precision score of 84.8%.

Next, we used the GitHub API (`:owner/:repo/commits/:id`) to extract the commit related information such as commit changed lines of code, modified files from the candidate fault-inducing commits. We eliminated the fault-inducing commits where the modified files are not the actual source code but are related to documentation, licence, test case, or git-related files shown in Listing 2 and obtained a list of 51,564 unique fault-inducing commits that modified a total of 56,003 unique files across all the studied projects. In Fig. 5, we report the distribution of fault-inducing commits and the duration (in months) to fix each of the identified fault for the fault-fix containing the issues reference (BR) and on-demand fault-fixing commits. We computed the duration as the difference in date of a given fault-fixing commit date and fault-inducing commit(s) date. The mean value was used for the case where more than one fault-inducing commits correspond to a single fault-fixing commit. In line with the work by Zhong and Su (2015), we observed that on average, the fault reported through the issue tracker has commits ranging from two to four, while the fault for which there is no issue reference (i.e., on-demand fault fixes) have only one commits on average, and they are fixed much faster than the fault from BR.

4.8. Correlation of technical debts and fault-inducing

This step aims to investigate how technical debts and fault-inducing changes are correlated. Examining the relation between them can show us the potential impact of technical debts. This information can also provide explanatory answers regarding the potential relationship between maintenance and reliability activities. To predict the occurrence of faults due to technical debts, we built multiple regression models on the set of technical debt metrics derived from the results of **RQ1**, which will be described in Section 5.1. We also used the number of fault-inducing commits (described in Section 4.7) as the dependent variable. Each data point is derived at the files level of a given project's snapshot. The metrics used in the regression model include:

- *File_ID:* A unique identifier is sequentially assigned to every new file added to the target project since the beginning of the development. We used this identifier to map the file history activities, such as a renamed file or deleted files. To assign the identifier, we first identified all code changes at the beginning of the project development and sequentially assigned a number (from 1) to all unique added files within the commit. We then checked for every new commit added to the project repository, and used the `git diff` command to detect any code modifications. For a renamed file, we mapped all file paths with the original file path and assigned the previous identifier to the file. For an added file, we gave a new sequential number. The *File_ID* was used to map the technical debts of the same files that may have been renamed.
- *File_size:* number of lines of code in a file as computed by SonarQube. This metric was used as the control variable when examining the relationship between technical debts and fault inducing occurrence.
- *Is_smelly:* A dummy value 1 if at least one smell was reported in the file, otherwise 0.
- *Is_erroneous* A dummy value 1 if at least one error was reported by SonarQube in the given file, otherwise 0.
- *Tag:* Number of different types of technical debts in a specified file. The full list of types is presented in Table 4 such as 'accessibility', 'brain-overload', 'clumsy', 'redundancy'.

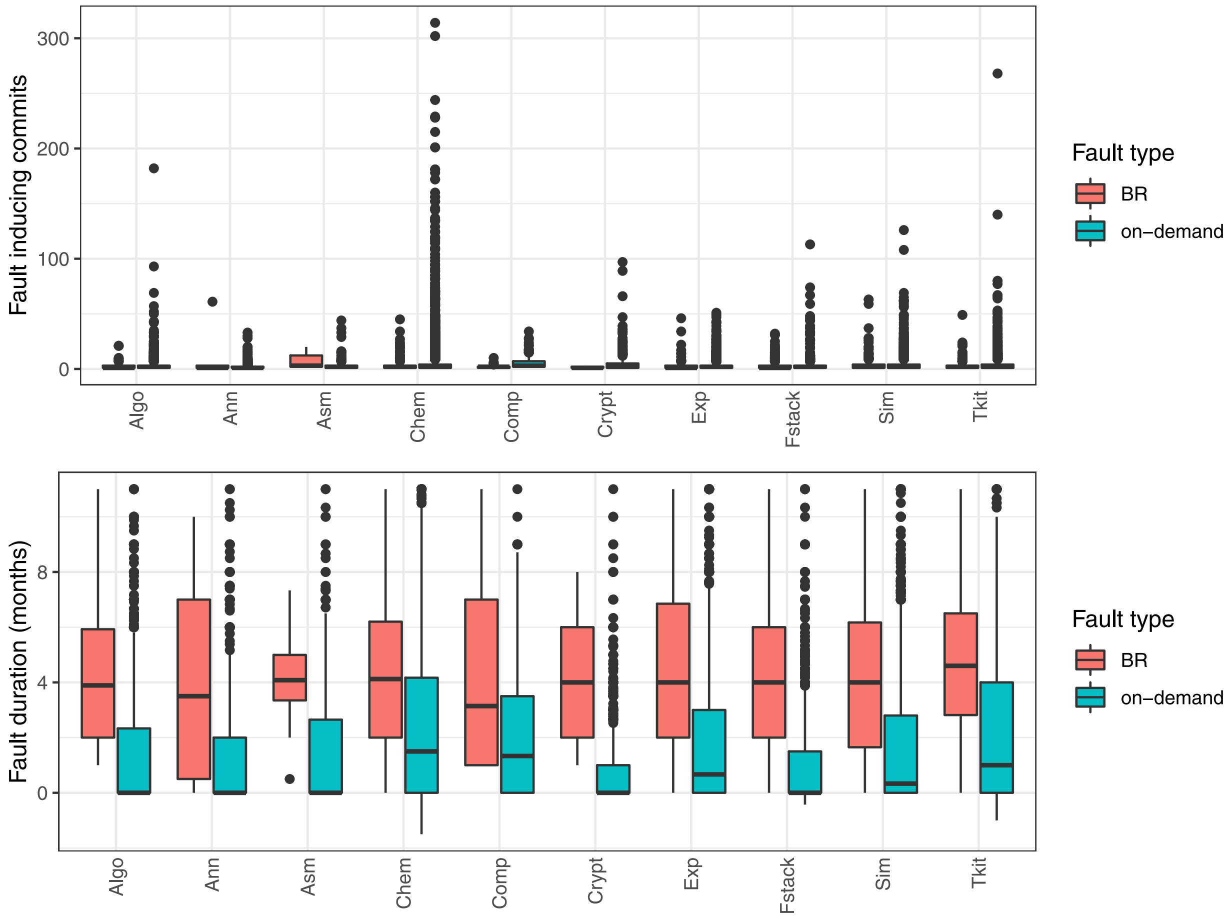


Fig. 5. The total number of fault-inducing commits (on top) and the duration (in months) to fix each of the faults identified using the fault-fix with issues reference (BR) and on-demand fault-fixing commits (in the bottom), computed using the date of fault-fixing and fault-inducing. **Algo**:Algorithms, **Ann**:Annealing, **Asm**:Assembly, **Chem**:Chemistry, *Comp*:Compiler, **Crypt**:Cryptography, **Exp**:Experimentation, **Fstack**:Full-stack library, **Sim**:Simulator, **Tkit**:ToolKit.

Approach:. We built Multiple Linear Regression (MLR) models for individual categories and a model for all combined quantum projects using the independent variables discussed above and the total number of fault-inducing commits (identified in Section 4.7) as the dependent variable. We used a separate model to examine how the technical debt metrics impact each category of the target projects. The combined category model was used to investigate the effect of each metric on the faults for all the target quantum projects.

Model building and evaluation:. We follow two steps suggested in previous studies (Shihab et al., 2010; Taba et al., 2013) to build our MLR models: removing the independent variables and analysis of multicollinearity.

The independent variable removal step eliminates the independent variables that are not statistically significant based on the dependent variable. For this case, we built a multiple regression model using the aforementioned metrics as independent variables and the number of fault-inducing commits as the dependent variable. We repeated this process and removed independent variables that have p-value ≥ 0.1, representing statistically non-significant variables.

Multicollinearity is a phenomenon that happens when two or more independent variables used in a regression model are highly correlated. This is an issue because it increases the variance of the regression coefficients, making them unstable. To check for multicollinearity in the models, we used the variance inflation factor (VIF). The VIF score of an independent variable represents how well the variable is explained by other independent variables. Therefore, the higher the value of VIF for an independent variable, the higher is the multicollinearity with that particular variable.

In this study, we follow previous works (Taba et al., 2013) and filter out independent variables for which VIF > 2.5. We used the `vif` function of the `car` package in R (Fox and Weisberg, 2011) to calculate VIF scores.

Next, we built the final model on the final set of independent variables. One of the parameters we used to assess MLR model's performance is the adjusted R^2 value instead of R^2. The adjusted R^2 is recommended to be more accurate for model assessment (Hastie et al., 2009) because it reflects the complexity of models. Besides, additional independent variables with lower explanatory power can directly reduce the value of R^2, which can be mitigated by using the adjusted R^2. We used the F-test (Wikipedia, 2020) to measure whether any of the independent variables are significant in the model. Another metric we used is Akaike's Information Criteria (AIC), which aims to add a penalty to the low importance variables of the model. Likewise, for the model selection, we considered the models with lower AIC values. We will discuss the results of this analysis in detail in Section 5.3 to answer **RQ3**.

5. Study results

This section presents the results of the analysis described in Section 4 answering our proposed questions **RQ1** through **RQ3**.

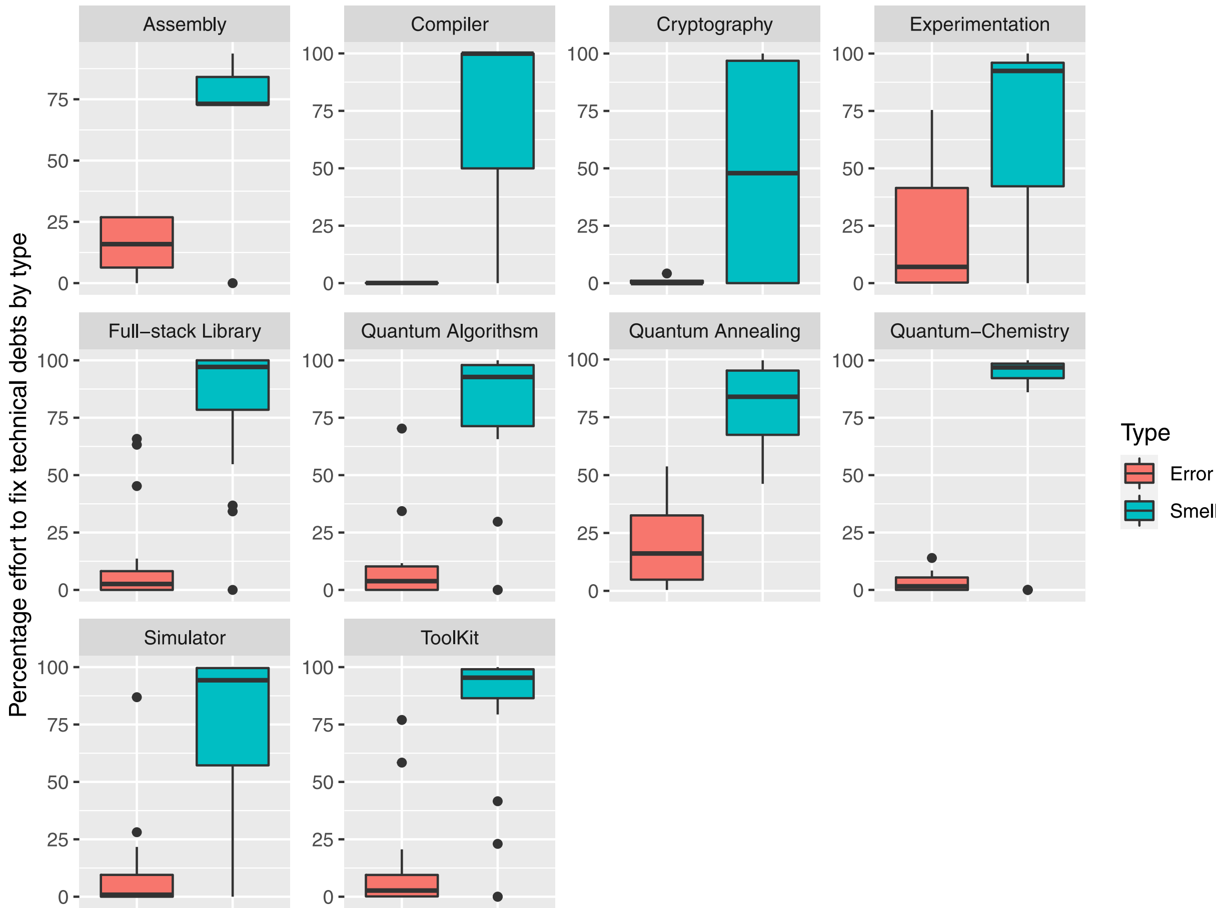


Fig. 6. Effort required to fix technical debts in terms of the debt types (coding errors and code smells).

5.1. RQ1: What are the characteristics of technical debts in quantum software?

We computed the average time to fix each debt type and separately debt severity across the snapshots of a given project. Next, we computed the resulting mean as a percentage across the debt types (i.e., coding error and code smell) and separately the percentage across debt severity (i.e., critical, major, minor, and blocker). Figs. 6 and 7 present the composition of technical debts by quantum software category. We show the percentage of the estimated time required to fix the technical debts (errors and code smells) and the severity assessed based on the total fixing effort in each of the project's snapshots in a quantum category.

As shown in Fig. 6, technical debts reported as code smells would require more fixing effort than errors (potential bugs). It implies that developers need more effort to fix maintenance-related issues (*e.g.,* code smells) than reliability-related issues (*e.g.,* coding errors). We observed a low value of the standard deviation (σ) across the target systems (from 2% to 25%), which indicates the consistency of required fixing effort in different quantum projects from different categories. Our results also show that coding errors in different categories do not require the same fixing effort. For example, errors in the categories of 'Quantum-Annealing', 'Experimentation', and 'Toolkit' require the most fixing effort while errors in the categories of 'Cryptography', 'Compiler', and 'Quantum-Chemistry' require the least fixing effort.

As indicated by Tan et al. (2020), Molnar and Motogna (2020) and Saika et al. (2016), technical debts can increase development overhead. Some of our findings can concretely explain this. For example, in `BBN-Q/QGL` (an experimental quantum application for domain-specific language embedded to specify pulse sequences), SonarQube detected about 2000 coding errors and 20,000 code smells, which respectively estimated to require about 42 days and 186 days to fix. Another example is that, in certain programming languages (such as Python), function names should follow a lowercase convention. Generally, the naming convention is considered necessary in a project with a shared team for effective collaboration. Renaming each of the functions in `BBN-Q/QGL` will take an average of two minutes.

We also observed multiple coding errors with the critical severity in the project `Quantomatic/pyzx` (a compiler project for rewriting quantum circuit and optimization). For example, a single file `pyzx/circuit.py` for representing a quantum circuit has up to 78 cognitive complexity. Cognitive complexity is a measure of how hard to understand the control flow in a function, and the high Cognitive Complexity will be challenging to maintain (Campbell, 2018). The maximum authorized cognitive complexity of the file is 15, indicating that the file needs about one hour to get fixed. In another error with the critical severity, the constructor method uses 12 parameters. A long parameter list indicates that the function or method is doing too many things, or that a new structure should be created to wrap the numerous parameters (SonarQube, 2021a) suggested that functions or methods should use less than 7 parameters to make the code maintainable. Refactoring each of these functions or methods requires about 20 min.

In Fig. 7, we further observed that the technical debts with the major and critical severity, require the most fixing effort, while debts with the blocker severity require the least fixing effort. Compared to other severity types, the fixing effort of critical debts has a higher standard deviation, which indicates that errors with this level of severity do not require a similar amount of time to

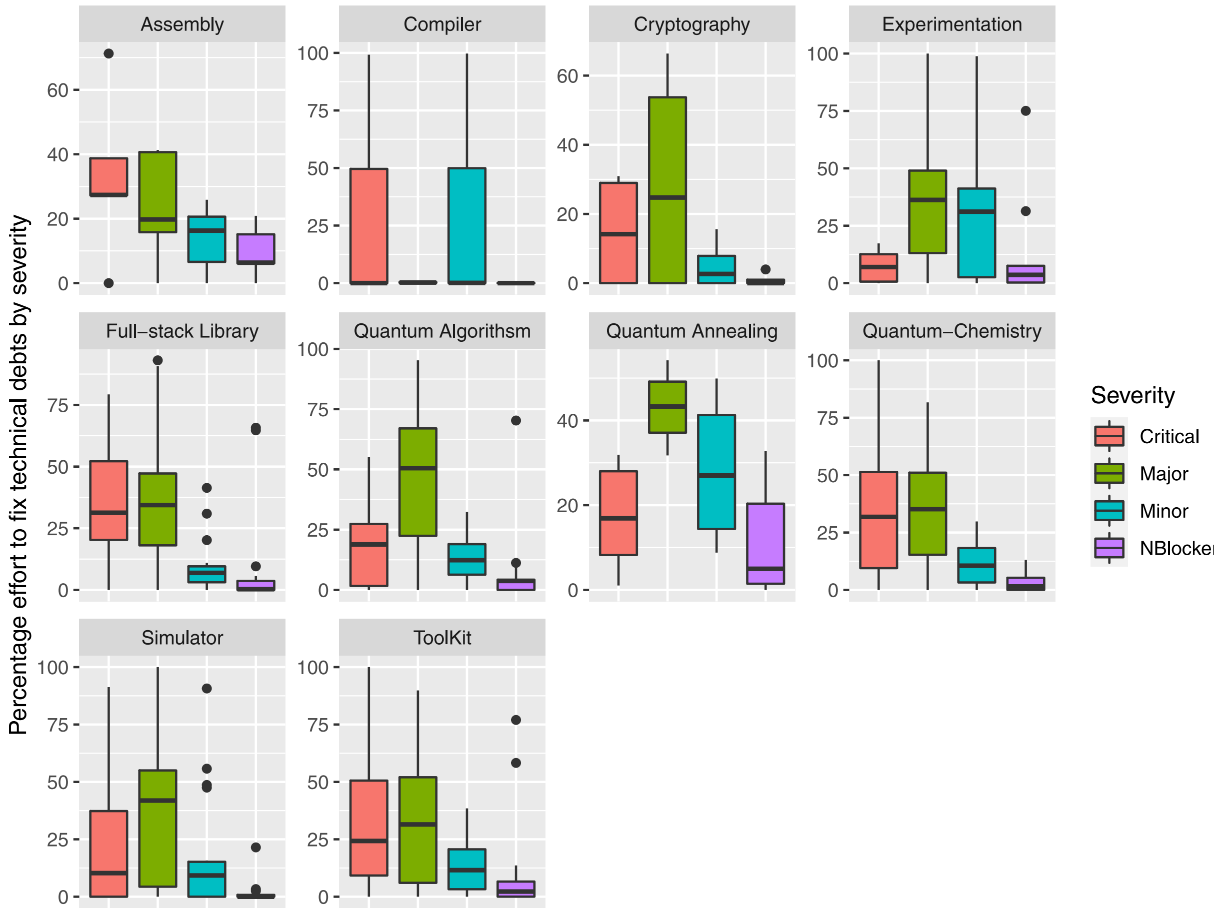


Fig. 7. Effort required to fix the technical debts in terms of types.

get fixed. In addition, we observed that debts in the categories of 'Compiler' and 'Full-stack Libraries' require the most fixing effort.

Overall, our findings are consistent with previous studies (Molnar and Motogna, 2020; Digkas et al., 2018; Marcilio et al., 2019) where people studied technical debts in traditional software. For instance, Digkas et al. (2018) studied technical debts in 57 software projects developed by the Apache ecosystem. They presented that more than 68% of identified technical debts were related to the major category. Marcilio et al. (2019) mined 426 different projects and reported that 68% of all types of technical debts belong to the major class. Molnar and Motogna (2020) carried out research on technical debts in three Java-based open-source projects. They showed that the technical debts with major severity account for 57.5% in FreeMind, 74.5% in jEdit, and 61.8% in TextGuitar. In other words, their result shows that a high percentage of technical debts have a major severity.

In Tables 3 and 4, we examined the types of technical debts in the target projects based on the set of rules and types defined by SonarQube (2021a). Table 3 shows the set of rules contributing to 80% and more of the overall technical debts in the target quantum categories. The highlighted rows indicate the debts classified as CRITICAL. From Table 4, we found the most prevalent types of debts: 'convention' (problem with coding convention such as formatting, naming, white-space), 'unused' (unused code, *e.g.,* commented out code or unused private variable), 'brain-overload' (related to cognitive complexity, there is too much code to keep in the head at once), 'design' (the design of the code is questionable, *e.g.,* duplicate string literals), 'cwe' (relates to a rule in the Common Weakness Enumeration CWE, 2021), and clumsy (unnecessary steps was used on something that should be straightforward) (SonarQube, 2021a). According to our findings in Table 4, the type 'code convention' dominates in at least 50% of the studied categories, followed by the high rate of 'unused code', 'design smells' and 'brain-overload'. We also observe that 'Quantum Chemistry' shows a higher percentage of code deprecation and design issues. In addition, a high percentage of code in the 'Compiler' and 'Full-stack Library' categories are characterized as confusing and suspicious.

In our manual investigation of source codes of two projects from 'Compiler' category (`Quantomatic/pyzx` and `QE-Lab/OpenQL`) we found that most of the technical debts were related to handling and generating graphs, benchmarks, unreachable code (due to jump statements), and unexpected expressions.

Also, we manually investigating the source code for the project `artiste-qb-net/qubiter` of Assembly category. We observed most of the code smells and coding errors detected (such as suspicious code, design, unused, brain-overload) are related to the source code for implementing in-memory storage of the circuits. In `artiste-qb-net/qubiter`, circuits are stored entirely in memory, essentially as Python lists of gates. Lists of this type can be sliced, combined, etc. When using such lists, it is easy to randomly access any gate of the circuit at will, for instance, when doing circuit optimizations (e.g., replacing the circuit with an equivalent shorter one). `artiste-qb-net/qubiter` uses a Python list of the lines stored as strings of the circuit's English file. For example, we observed multiple empty blocks of codes detected as 'suspicious' in the function for generating the Python list of gates of Hermitian conjugate. These features are particular in quantum software, while in traditional software, the most frequent technical debts are related to different groups. Marcilio et al. (2019) explained that the problems regarding packages and exception handling contain the biggest proportion of technical

Table 3
Rules generating about 80% or more of technical debts, indicating the mean percentage of occurrence in overall application snapshots, severity, type of technical debts, and the descriptions of the rule (as indicated by SonarQube). The highlighted rows indicate the technical debts with the CRITICAL severity.

Rule(severity)	Cryp	Exp	Ann	Fstack	Tkit	Chem	Comp	Asm	Alg	Sim	Tag SonarQube (2021a)	Example descriptions
S117(minor)	17.5	51.73	29.85	13.01	18.42	11.55	2.07	26.64	25.91	36.7	convention	Parameter naming convention
S1542(major)	32.98	6.09	6.59	6.32	6	8.15	4.17	10.54	4.35	14.01	convention	Function naming convention
S3776(critical)	8.92	1.98	5.13	6.43	4.48	8.64	21.88	8.64	8.36	1.96	brain-overload	Cognitive Complexity of function/ method should not be too long
S1192(critical)	7.85	1.99	2.79	8.61	14.12	13.69	2.46	2.88	10.56	7.19	design	Define a constant instead of duplicating this literal
S1481(minor)	1.77	1.99	12.65	5.89	7.98	5.56	6.46	7.59	6.72	8.25	unused	Remove this unused declaration should be removed
S125(major)	0.61	4.33	4.69	1.85	5.15	10.67	19.12	6.65	5.38	4.88	unused	Remove this commented out code
PrintStatement Usage(major)	0.54	1.34	4.5	6.04	3.9	18.55	0	0.67	11.75	1.33	python3,obsolete	Replace print statement by built-in function.
S3827(blocker)	0.16	3.95	6.57	1.12	1.47	1.42	0.94	6.49	8.71	0.81	undefined usage.	
S1117(major)	0.04	0	0	8.26	1.05	0.08	13.95	0.56	0.06	1.58	confusing,pitfall, suspicious	Two or more field should not be declared with same name
S5754(critical)	2.01	0.9	0.45	4.43	7.12	1.59	3.64	1.88	0.58	1.94	confusing,unpredictable, error-handling	Specify an exception class to catch or reraise the exception
S100(minor)	2.46	3.8	3.27	4.32	1.66	1.85	0.44	1.54	0.31	0.97	convention	Method naming convention.
S112(major)	4.19	0.61	0.77	1.46	3.32	1.52	5.61	1.6	0.64	0.3	error-handling,cwe	Replace this generic exception class with a more specific one.
S1827(major)	0	0.88	0.96	0.21	3.09	0.74	0	1.55	0.57	6.55	user-experience,html5, obsolete	Remove this deprecated attribute
S107(major)	0.32	1.04	2	1.48	2.77	1.37	1.15	0.76	2.4	0.82	brain-overload	Parameter list of a function should not be greater than the 7 authorized.
S101(minor)	0.23	0.65	1.94	2.13	1.06	1.01	1.57	1.28	1.2	0.22	convention	Class naming convention
S1066(major)	1.04	0.38	0.41	2.28	1.36	1.37	1.66	0.68	1.82	0.25	clumsy	Merge this if statement with the enclosing one.
S116(minor)	2	2.93	0.19	1.0	2.75	0.84	0.27	0.01	0.21	0.44	convention	Field naming convention
BoldItalicTags Check(minor)	0.02	0.47	0.91	2.64	0.84	0.2	0.34	3.27	0.99	0.81	accessibility	Replace this <i> tag by <em>.
S1186(critical)	1.38	0.39	0.18	2.32	1.48	0.33	0.38	0.57	0.34	2.86	suspicious	Add a nested comment explaining why this method is empty, or complete the implementation.
	84.02	**85.45**	**83.85**	**80.0**	**88.02**	**89.13**	**86.11**	**83.8**	**90.86**	**91.87**		

debts in their studied projects. Digkas et al. (2018) also reported that the most frequent technical debts are related to resource management, null pointer, and exception handling problems.

We also observed that the files containing debts have high cognitive complexity due to a large number of code lines, loops, and if-else statements. For example, a function used for computing the echelon form of a matrix has the cognitive complexity of 155, which exceeds the maximum allowed cognitive complexity of the function (*i.e.,* 140). Moreover, we found 86.6% of the code duplication on 70,000 lines in `Quantomatic/pyzx` and 88.9% code duplication on 1.3M lines in `QE-Lab/OpenQL`.

We will further discuss the implication of our findings in Section 6.

> On the one hand, more than half of technical debts in all kinds of software systems (quantum and non-quantum) are classified as major from the severity point of view. On the other hand, the most frequent types of technical debts are different in various software types. With respect to this fact, we found that the most frequent technical debts in quantum computing software systems are related to the 'code convention', 'design issue', 'brain-overload', and 'error-handling' problems.

Table 4
Types of technical debts that contribute to greater than 80% of the overall technical debts across different categories of the quantum projects (*s* for code smells and *e* for coding errors).

Tag SonarQube (2021a)	Type	Cryp	Exp	Ann	Fstack	Tkit	Chem	Comp	Asm	Alg	Sim
convention	s, e	48.31	56.9	36.92	18.49	21.66	17.65	6.14	37.99	26.44	40.71
unused	s, e	2.62	6.74	15.62	8.03	10.63	13.52	18.87	14.61	12.45	11.56
brain-overload	s	7.62	2.64	6.25	5.45	5.16	7.48	15.67	8.14	8.96	2.16
design	s	6.44	1.73	2.48	6.13	10.05	10.28	1.96	2.63	8.93	5.8
obsolete	s, e	0.45	4.12	6.38	4.83	6.82	14.75	0	3.09	10.9	6.19
cwe	s, e	3.77	2.61	1.33	3.45	3.95	3.04	5.5	2.71	3.08	0.54
confusing	s, e	3.1	1.16	0.94	11.28	6.38	1.63	13.74	2.87	1.13	3.53
error-handling	s, e	5.08	1.42	1.09	4.48	7.46	2.77	6.33	2.95	1.08	2.43
accessibility	s, e	6.93	0.74	1.02	3.83	2.33	3.17	0.47	4.31	1.05	3.18
suspicious	s, e	2.65	1.07	1.09	9.2	2.33	1.19	11.03	1.92	1.24	3.78
python3	s, e	0.45	1.18	3.96	4.67	2.78	14.15	0	0.57	9.91	1.04
user-experience	s, e	0	3.03	2.44	1.49	4.05	0.69	0	2.78	1.21	5.43
html5	s, e	0	2.95	2.44	0.16	4.01	0.66	0	2.52	1.2	5.15
unpredictable	s	1.64	0.85	0.4	3.3	5.04	1.35	2.51	1.59	0.48	2.1
wcag2-a	s, e	6.92	0.33	0.21	2.02	1.69	3.02	0.24	1.53	0.23	2.52
clumsy	s	1.59	0.43	0.6	1.96	1.65	1.51	2.45	1.15	1.61	0.34
Total		**97.57**	**87.9**	**83.17**	**88.77**	**95.99**	**96.86**	**84.91**	**91.36**	**89.9**	**96.46**

5.2. RQ2: How do technical debts evolve over time?

This section discusses how technical debts evolve in the target quantum projects. From the 10 identified quantum software categories, we selected the most representative projects in each categories, *i.e.,* 'Assembly', 'Quantum-Annealing', 'Experimental', 'Full-stack Library', 'Quantum-chemistry', 'Toolkit', 'Simulator', 'Quantum-Algorithms', 'Cryptography', and 'Compiler'. We discriminated the list of file added or modified in the current snapshot from the other files using the git diff command and computed the debts in these files normalized using the KLOC of the files. Figs. 8 (8a to 8t) provides the general summary of how the technical debts (in terms of code errors on the left and code smells on the right) evolve in the studied categories. Each line represents the evolution of technical debt ratio (TDR, described in Section 4.4.1) on the *y*-axis and the number of days based on 90 days intervals (snapshots) on the *x*-axis, for a single project. In the following, we report our results from manually analyzing two projects in each of the categories and discuss the evolution of technical debts in detail.

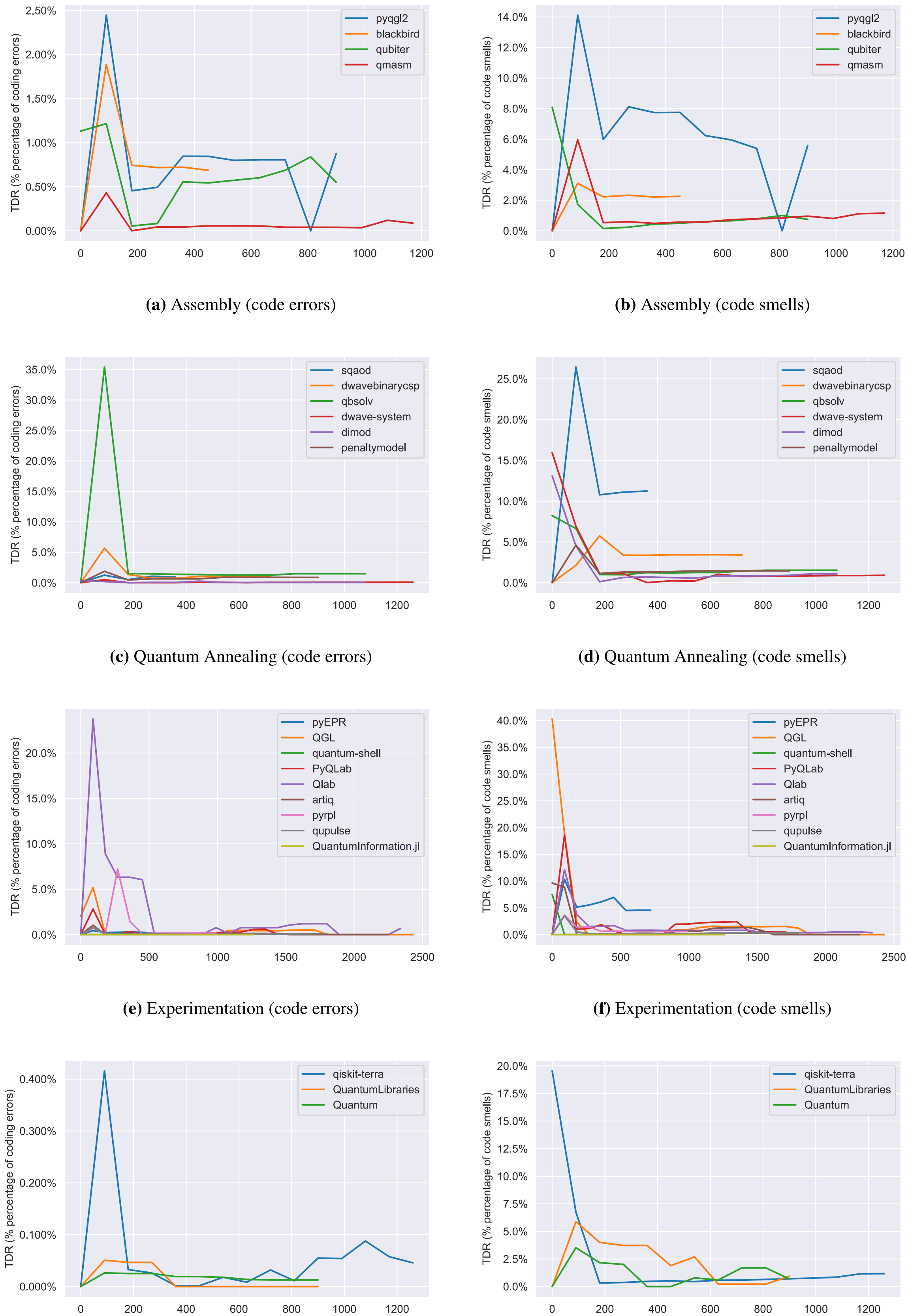
Figs. 9 (9a to 9t) illustrates how new technical debts were added to the projects over time across the studied snapshots. The technical debt in the first snapshot of each application is new; the divergence from the horizontal line indicates the supplementary debt that has been introduced or removed or both. Each subfigure is scaled to indicate the technical debts ratio (TDR) on the left and KLOC on the right. The KLOC allows us to identify the snapshots where key development activities took place, such as new code added or deleted and how these activities are related to technical debts.

In the figure, we observed that there is a general sharp raise of TDR at the initial phase of the software development (more than 10% of corresponding to SQALE maintainability rating from *B* and *C*), which later gradually decreased as the code base matured in most of the target projects. Also, according to Fig. 9, we found that the introduction of technical debts is highly related to the project size (KLOC). For the Assembly project `artiste-qb-net/qubiter`, we can see that most of the code was added within the first two snapshots (corresponding to the development time of up to 180 days), in which most new added debts are related to code smells. We observed fewer code changes in the later snapshots of the project. These changes have little relationship with technical debts. Similarly, we observed a sharp rise of technical debts in the `BBN-Q/pyqgl2` project.

For the case of the Quantum Annealing project `dwavesystems/qbsolv`, the majority of the codebase was added between the first 90 and 180 days. The added code contains more coding errors than code smells. The number of technical debts remains stable in the later snapshots of the project. Similar trends are observed in the Full-stack library project `Blueqat/Blueqat` where more coding errors were detected mainly in the initial snapshot of the projects compared to the later snapshots.

Higher debts and the sharp rise of code lines at the initial snapshots may be due to the fact that most new files are added at the initial phase of the development. When a project becomes mature, fewer added and deleted lines are observed, which can be associated with minor or modification changes. This was also observed in some of the projects such as `artiste-qb-net/qubiter` where a single commit `#e3f5539` added 71 new files with over 8800 new lines of code in the first snapshot. With a manual investigation on 30 of these files, we found both coding errors and code smells existing in the files. Some of the files were later refactored, which implies that developers realized the technical debts and removed them. Tsoukalas et al. (2020) carried out research on 15 open-source projects and showed that complexity and LOC are two of the most significant indicators of technical debts. Siavvas et al. (2020) also studied 150 open-source software projects and reported that cyclomatic complexity of code is one of the key indicators of technical debts.

We observed a similar relation for quantum projects. We observed a high positive correlation between file size and the amount of technical debts. Table 5 presents the results of our correlation analysis (using the Spearman rank correlation test Zar, 2005) between file size (LOC) and reported new technical debts across the snapshots of the randomly selected quantum projects from Fig. 9. The low values of standard deviation indicate that the results are consistent across the project snapshots. We also found that at least half of the reported technical debts are contributed by the top 20% of files (according to LOC) in all the studied snapshots, while the bottom 20% of files only introduced a few technical debts ($\leq$5%). Fig. 10 illustrates the percentage of unique files contributing to at least 80% of the technical debts across the snapshots of our studied projects. Our results confirms the Pareto's principle (80–20 rule) (Dunford et al., 2014) and is in line with the previous studies (Walkinshaw and Minku, 2018; Molnar and Motogna, 2020) on software defects and technical debts in traditional software. Walkinshaw and Minku (2018) studied 100 open-source projects and reported that 80.5% of fixes are related to just 20% of the files in all projects. Besides, they identified that near to 73% of LOC of the top 20% files are involved in the top 80% of fixes.



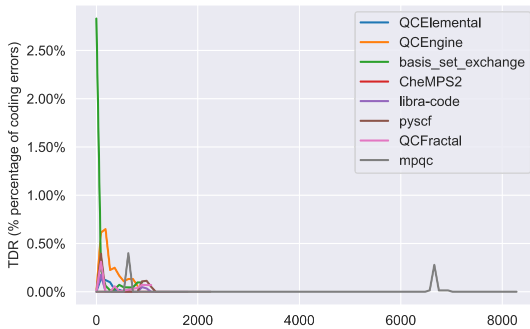
(a) Assembly (code errors) **(b)** Assembly (code smells)

(c) Quantum Annealing (code errors) **(d)** Quantum Annealing (code smells)

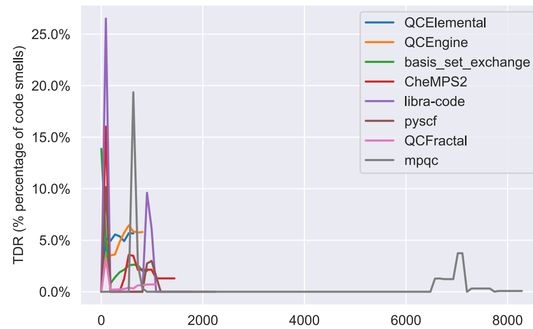
(e) Experimentation (code errors) **(f)** Experimentation (code smells)

(g) Full-stack Library (code errors) **(h)** Full-stack Library (code smells)

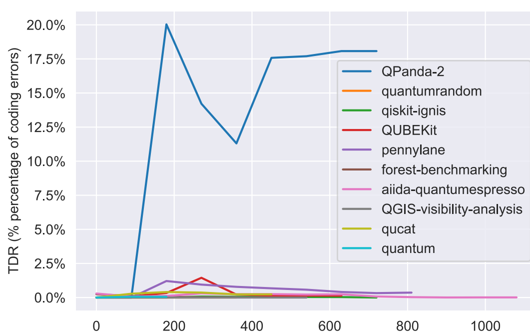
Fig. 8. How the technical debts (coding errors on the left and code smells on the right) evolve for ten randomly selected projects in each of the eight studied categories with high rate of technical debts. Each line represents the evolution of technical debt ratio (TDR, described in Section 4.4.1) on the y-axis and the number of days based on 90 days interval (snapshots) on the x-axis, for a single project.



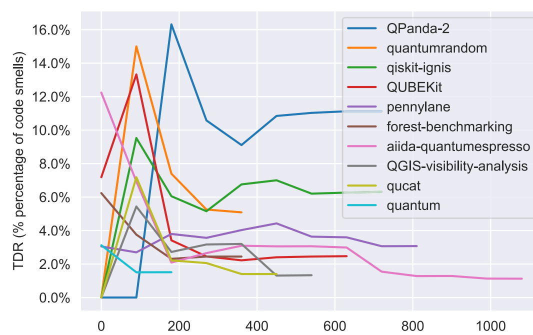
(i) Quantum Chemistry (code errors)



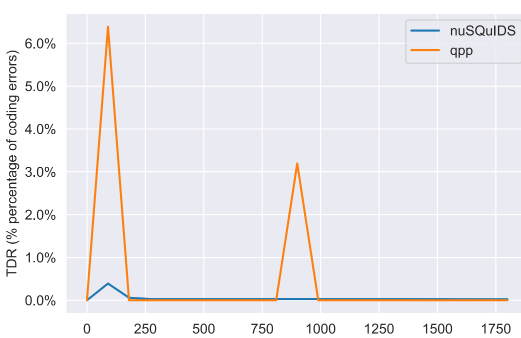
(j) Quantum Chemistry (code smells)



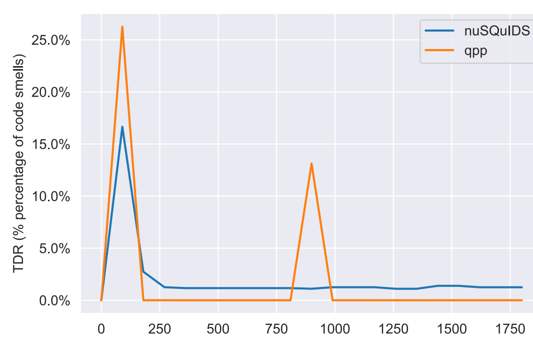
(k) ToolKit (code errors)



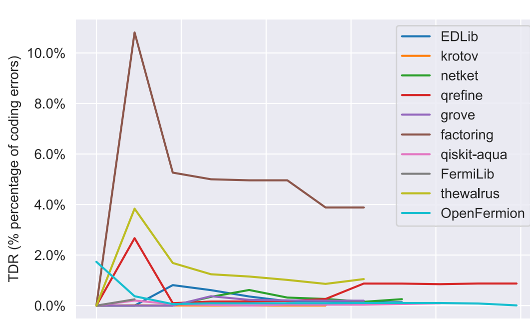
(l) ToolKit (code smells)



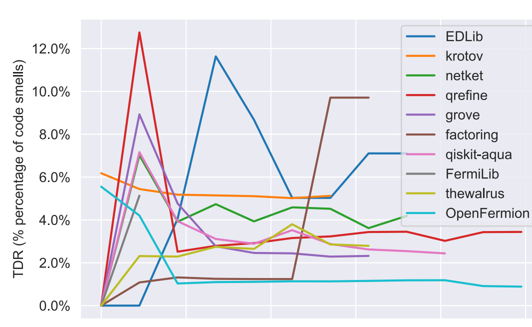
(m) Simulator (code errors)



(n) Simulator (code smells)

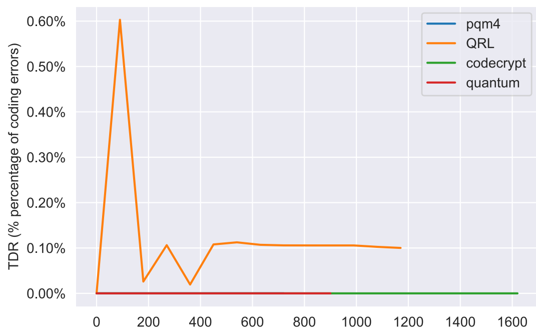


(o) Quantum Algorithms (code errors)

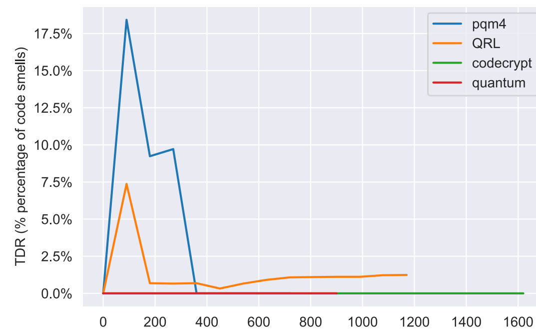


(p) Quantum Algorithms (code smells)

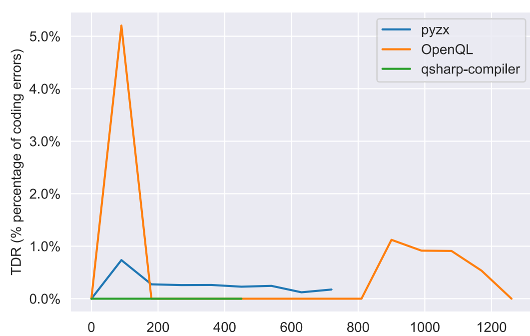
Fig. 8. (*continued*).



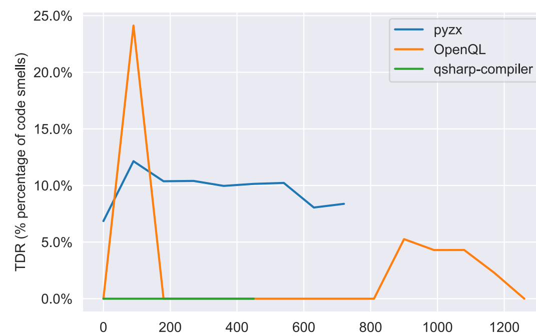
(q) Cryptography (code errors)



(r) Cryptography (code smells)



(s) Compiler (code errors)



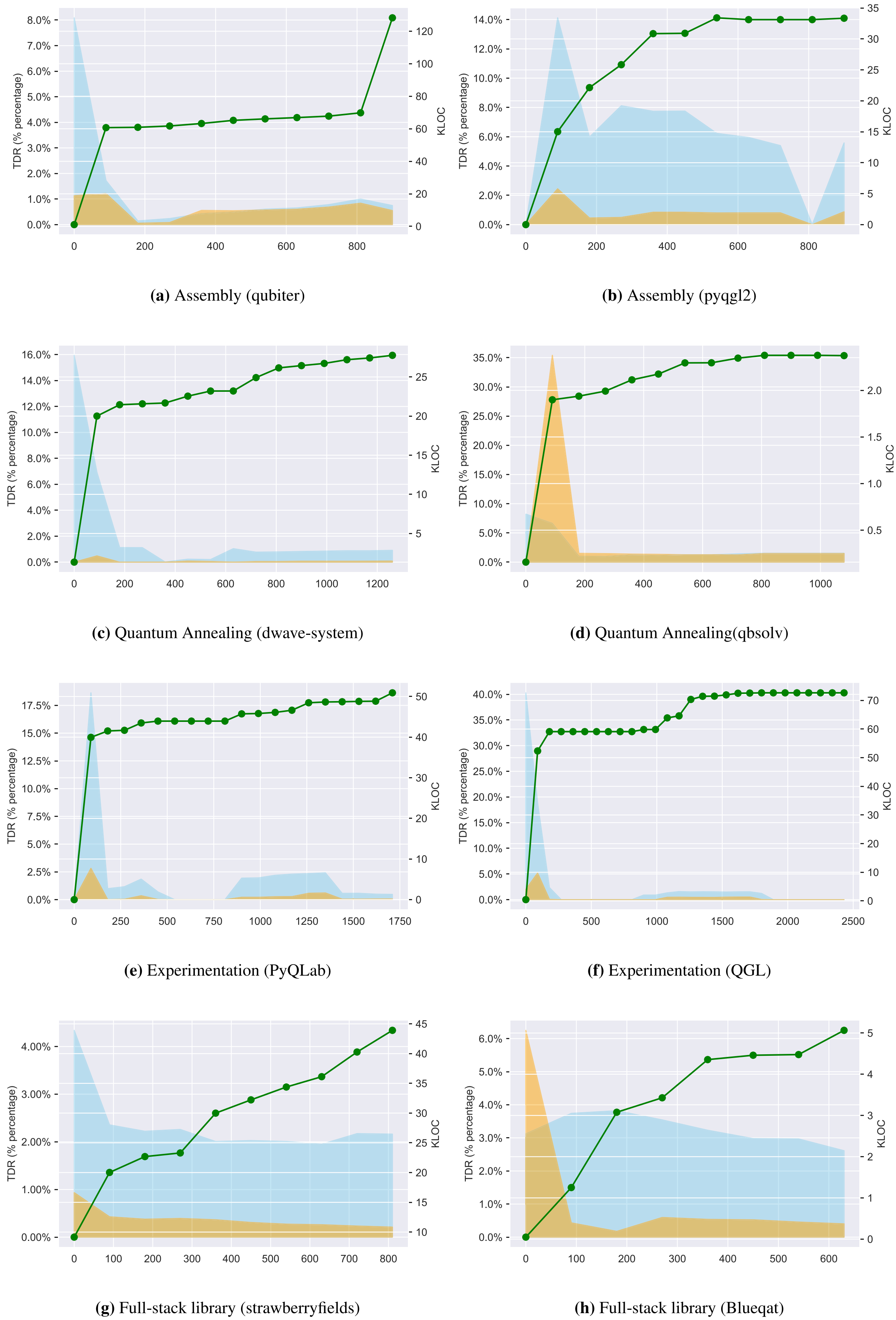
(t) Compiler (code smells)

Fig. 8. (*continued*).

Table 5
The Spearman rank correlation coefficient that evaluates the correlations between the total technical debts in a file and the lines of code of the file. We calculated the correlation coefficient for two randomly selected quantum projects from each quantum categories in Fig. 9.

Category	Target project	Correlation	p-value
Assembly	`qubiter`	0.622	***
	`pyqgl2`	0.895	***
Quantum-Annealing	`dwave-system`	0.893	***
	`qbsolv`	0.879	***
Experimental	`PyQLab`	0.940	**
	`QGL`	0.936	**
Full-stack Library	`strawberryfields`	0.895	***
	`Bluegat`	0.902	***
Quantum-Chemistry	`QCFractal`	0.903	***
	`CheMPS2`	0.883	***
Toolkit	`qucat`	0.820	***
	`bloomberg/quantum`	0.949	***
Simulator	`SimulaQron`	0.855	***
	`Quirk`	0.584	***
Quantum-Algorithms	`grove`	0.838	***
	`qiskit-aqua`	0.860	***
Cryptography	`pqm4`	0.943	*
	`QRL`	0.551	***
Compiler	`OpenQL`	0.983	***
	`pyzx`	0.879	***

*Statistical significance denoted as <0.05.
**Statistical significance denoted as <0.01.
***Statistical significance denoted as <0.001.



(a) Assembly (qubiter)

(b) Assembly (pyqgl2)

(c) Quantum Annealing (dwave-system)

(d) Quantum Annealing(qbsolv)

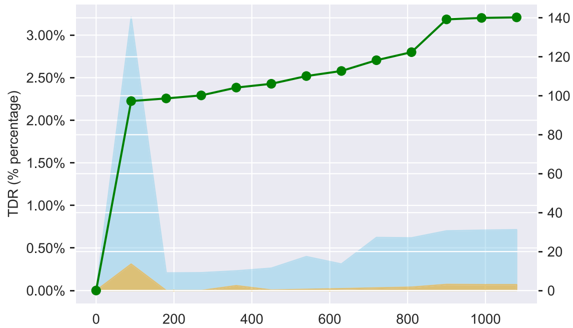
(e) Experimentation (PyQLab)

(f) Experimentation (QGL)

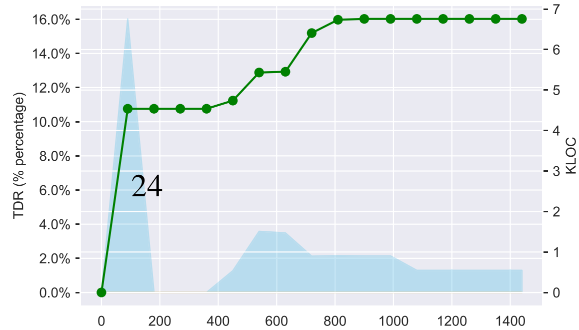
(g) Full-stack library (strawberryfields)

(h) Full-stack library (Blueqat)

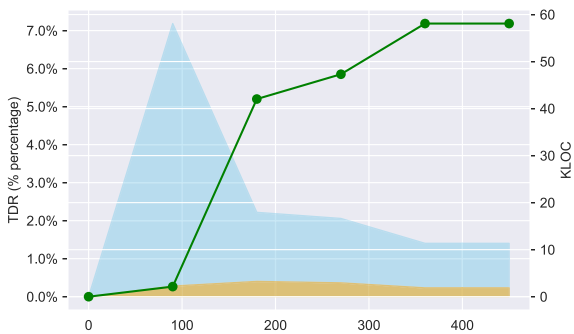
Fig. 9. How the technical debts (coding errors and code smells) evolve with development activities (the added and deleted lines of code (KLOC)) for two randomly selected projects in each of the eight studied categories with high rate of technical debts. We show the technical debt ratio (TDR, described in Section 4.4.1) on the left side of y-axis and the KLOC on the right side (calculated from LOC returned by SonarQube) and the number of days based on 90 days interval (snapshots) on the x-axis.



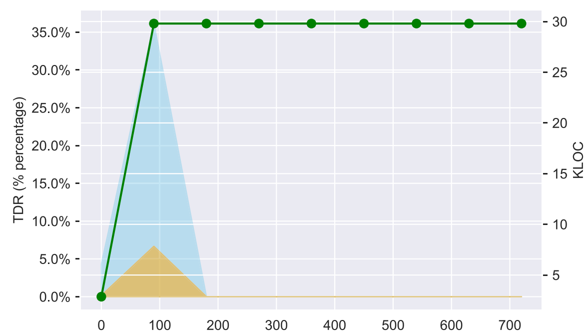
(i) Quantum-chemistry (QCFractal)



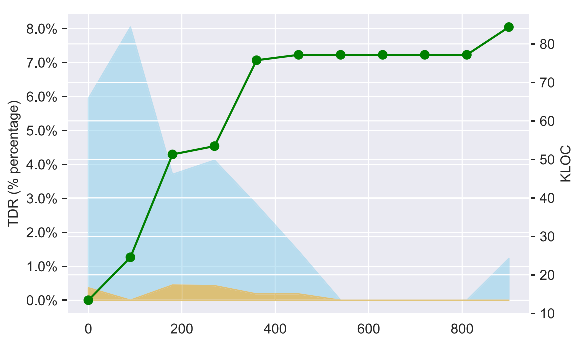
(j) Quantum-chemistry (CheMPS2)



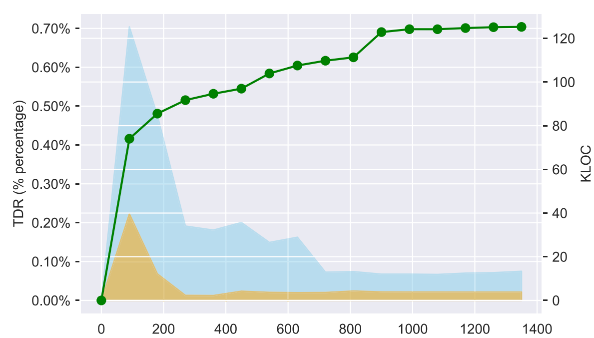
(k) Toolkit (qucat)



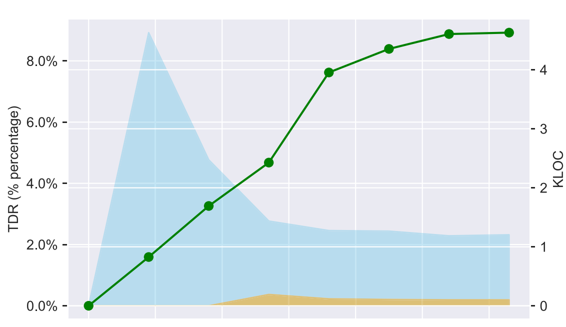
(l) Toolkit (bloomberg)



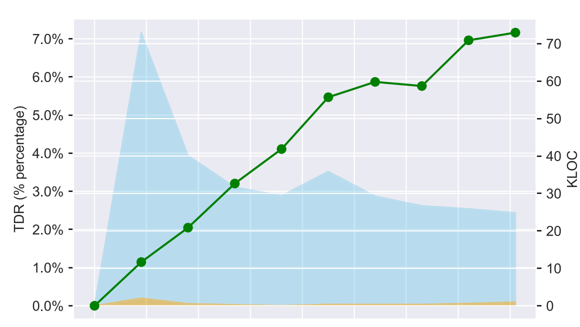
(m) Simulator (SimulaQron)



(n) Simulator (Quirk)

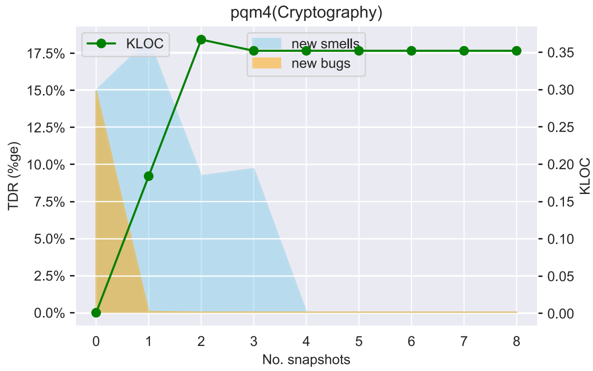


(o) Quantum algorithm (grove)

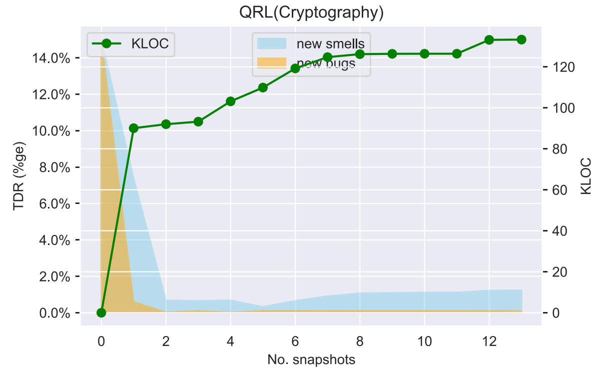


(p) Quantum algorithm (qiskit-aqua)

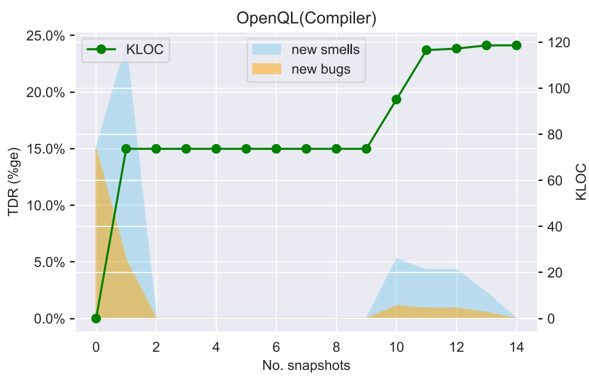
Fig. 9. (*continued*).



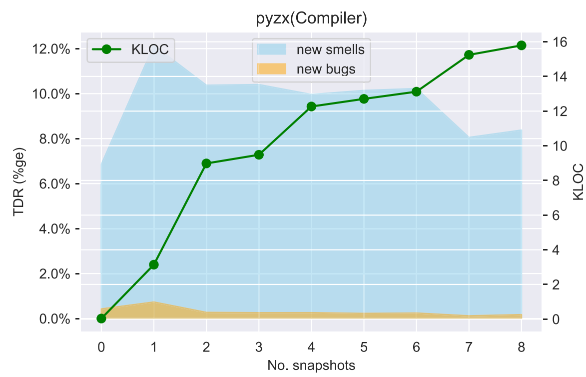
(q) Cryptography (pqm4)



(r) Cryptography (QRL)



(s) Compiler (OpenQL)



(t) Compiler (pyzx)

Fig. 9. (*continued*).

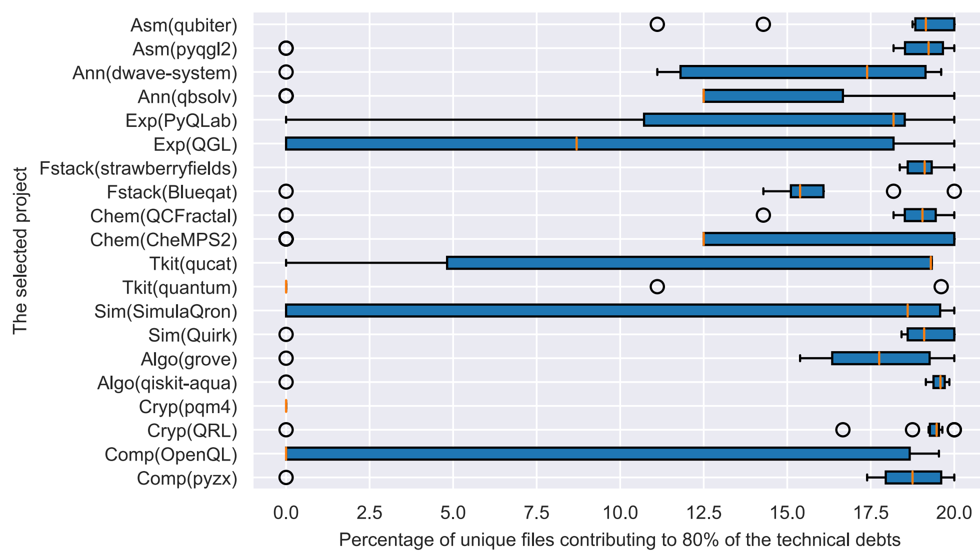


Fig. 10. Percentage of unique files contributing to 80% of the overall technical debts across the snapshots of the selected projects.

Technical debts tend to be added in the initial versions of a project in quantum computing software systems. Furthermore, similar to other types of software systems, we show that LOC is a key indicator of the occurrence of technical debts in quantum computing software systems.

5.3. RQ3: What is the relationship between technical debts and faults?

In Section 4.8, we described the steps used to examine the relationship between technical debts and faults introduced, as well as metrics that are related to the fault-inducing changes. This section discusses the results of the analysis of Section 4.8.

Table 6
p-values for statistical significance and the coefficient for the Multiple Linear Regression (MLR) model based on all quantum projects (ALL-QT). The model uses the fault-inducing commits as dependent variable and the list of independent variables derived from technical debts types (as described in Section 4.8).

coef/p-value	ALL-QT
is_smelly	0.0011***
is_erroneous	0.0014***
project	0.0086***
accessibility	[a]
brain-overload	0.0459***
design	0.0001**
unused	[a]
convention	6.532e−05*
cwe	[a]
redundant	0.0027***
confusing	[a]
error-handling	0.0069**
obsolete	[a]
user-experience	[a]
suspicious	[a]
Adj. R^2	0.059
Prob (F-stat)	0.00
AIC	−8.418e+05

† Variable removed during VIF analysis.
*Statistical significance denoted as <0.05.
**Statistical significance denoted as <0.01.
***Statistical significance denoted as <0.001.
[a]Variable removed during stepwise selection criteria.

Table 7
p-values for statistical significance and the coefficient for the MLR model based on individual quantum categories. The model uses the number of fault-inducing commits as dependent variable and the list of independent variables derived from technical debts types.

coef/p-value	Fstack	Tkit	Exp	Sim	Crypt	Comp	Alg	Chem	Asm	Ann
is_smelly	0.0541***	0.0009***	0.0021***	0.0875***	0.0002***	0.0417***	0.0194***	0.0559***	0.0072***	0.0088**
is_erroneous	0.1217***	0.0012***	0.0042***	[b]	0.0003***	0.0245***	[b]	0.1372***	0.0084*	0.0134***
project	[b]	0.0004***	[b]	[b]	0.0163**	[b]	[a]	2.9787***	[b]	0.4092*
accessibility	[b]	[b]	[a]	[b]	[a]	[a]	[a]	[b]	[b]	[a]
brain-overload	0.1245***	[a]	0.5711***	[a]	[a]	[a]	0.4022***	[a]	0.0070**	0.3344***
design	0.3402***	0.0084***	0.0050***	1.4217***	0.0003*	[a]	0.0342**	[b]	[b]	[b]
unused	[b]	[b]	0.0083***	0.1842***	[a]	[a]	[b]	0.2440**	0.0003***	[b]
convention	0.3238***	0.0006***	[b]	0.0310***	[a]	[b]	0.0246***	[b]	[a]	[b]
cwe	[b]	0.0209***	0.0713**	0.5695***	3.384e−05**	[a]	[b]	[b]	[a]	[b]
redundant	[b]	0.0179***	[b]	2.9021***	[a]	0.0609*	0.3671*	0.2996*	[b]	[b]
confusing	[a]	0.0861***	0.2705***	0.5060**	[a]	[a]	0.0270***	[b]	[b]	0.0601*
error-handling	[b]	[b]	0.0047***	0.0587**	0.0005**	[b]	0.1018**	0.0250**	0.0203*	[b]
obsolete	[a]	[b]	[a]	[a]	[a]	[a]	[a]	[a]	[a]	[a]
user-experience	[b]	[b]	[a]	[b]	[a]	[a]	[b]	[b]	[b]	[a]
suspicious	0.3689***	[b]	0.0023**	[b]	0.0010***	[a]	[b]	[b]	[b]	[b]
Adj. R^2	0.148	0.170	0.285	0.193	0.553	0.110	0.213	0.027	0.358	0.174
Prob (F-stat)	2.37e−267	0.00	3.79e−242	0.00	0.00	1.99e−69	3.76e−243	2.04e−275	2.09e−44	4.35e−34
AIC:	−1.598e+04	−9.974e+04	−2.314e+04	−1.043e+04	−5.376e+04	−1.291e+04	−2.257e+04	−5.579e+04	−3143.	−5434.

*Statistical significance denoted as <0.05.
**Statistical significance denoted as <0.01.
***Statistical significance denoted as <0.001.
[a]Variable removed during VIF analysis.
[b]Variable removed during stepwise criteria.

Tables 6 and 7 presents the Multiple Linear Regression (MLR) model fitting results showing the relationship between technical debt as dependent variables and the fault-inducing commits as the independent variable. The dependent variables *is_smelly* and *is_errorneous* indicate if a given file contains at least one technical debt of type code smell and code error, respectively. The rest of the independent variables are the total counts of different technical debts by tags (tags presented in Table 4) in a given file across the project's snapshots.

Table 6 reports the MLR results after fitting the model on all the quantum projects as one dataset (*i.e.*, the debts reported for all the target projects as independent variables and the total number fault-inducing commits as dependent variable at file level). In Table 7, we present the MLR fit result for the 10 individual quantum categories. We only reported the MLR model results for the p-value that is less and equal to 0.05 (i.e., p-value ≤ 0.05), the positive coefficient for statistically significant independent variables, and the variance inflation factor (VIF) that is less and equal to 2.5. The positive coefficient indicates the positive correlation between the independent variables (number of technical debts) and the dependent variable (number of fault-inducing commits).

From Tables 6 and 7, we observed a statistically significant correlation between code smells and the number of fault-inducing changes, which is shown by the positive coefficient values of 'is_smelly' in all the categories of studied projects. The high positive coefficient of 'is_smelly' for the quantum category 'Full-stack Library', 'Simulator', 'Quantum Chemistry', 'Compiler', and 'Quantum Algorithm' also shows that there is a particularly high correlation between the code smells and the number of fault-inducing commits in the respective target quantum software. In addition, we found statistically significant correlation between the coding errors and the number of fault-inducing commits in most (80%) of the target quantum categories. In particular, the coding Error shows a higher positive correlation on the fault introduced in the 'Full-stack Library', and 'Quantum Chemistry' categories.

Moreover, Table 7 presents the correlation of technical debts based on the tag and the number of fault-inducing commits. Our results show that the tags 'design', 'redundant', 'error-handling', and 'brain-overload' have a statistically significant correlation with the number of fault-inducing commits in most (≥50%) of the quantum categories. This result suggests that the tags 'design', 'redundant', 'error-handling', and 'brain-overload' would help predict the occurrences of fault-inducing commits in quantum software systems. The design technical debt has higher positive correlation to fault-inducing in Full-stack libraries, quantum simulators and Toolkit categories, 'brain-overload' shows a higher positive correlation with the number of fault-inducing commits in the quantum categories in the order 'Full-stack Library', 'Quantum-Annealing', 'Quantum-algorithms' and 'Assembly'. Also, we observed a positive correlation between the project and the fault-inducing in the quantum categories Quantum-Chemistry, Quantum Annealing, Cryptography and Toolkit.

Comparing the technical debts tags across the quantum categories, 'brain-overload' shows a higher positive correlation with the number of fault-inducing commits in the quantum categories in the order 'Experimentation', 'Quantum-algorithms', 'Quantum-Annealing', 'Full-stack Library', and 'Assembly'. Similarly, the design related technical debt has higher positive correlation to fault-inducing in 'Full-stack Library'. Moreover, the technical debt of tag 'redundant' introduces more faults in the categories 'Simulator', 'Quantum-Chemistry', and 'Compiler'. Finally, in Table 7 we observed that most of the tags are statistically significantly correlated with the number of fault-inducing commits in the quantum categories of 'Experimentation', 'Simulator', 'Quantum Algorithms', and Toolkit, whereas only fewer (≤20%) of the tags are correlated with the fault-inducing commits in the categories 'Compiler' and 'Quantum Annealing'. This implies that these technical debt tags can be included to predict the number of fault-inducing commits in the respective quantum categories.

Previous studies also reviewed the relationship between technical debts and types of faults in different types of software systems. Digkas et al. (2017) carried out a research on 66 open-source java based software projects developed by the Apache software foundation. They showed that 'literal duplicate' is the most frequent type of technical debts. They also observed that this type of technical debts are equally distributed among their analyzed systems. Moreover, they revealed that 'using diamond operation' and 'code comment-out' are the least equally distributed technical debts among their studied software systems. Tan et al. (2020) studied 44 open-source software systems based on Python belonging to the Apache software foundation. They indicated that 'defining docstring', 'too long lines' and 'undefined variables' are the top three types of technical debts identified in those projects. Moreover, they mentioned that 'empty nested block' and 'too many line of code in a file' are the least frequent types of technical debt.

> As expected, the correlation between technical debts and the number of faults induced are different based on the nature of software systems. On the other hand, the technical debts of types 'design', 'redundant', 'error-handling' and 'brain-overload' can be a good indicator to predict the number of fault-inducing commits in quantum software since they have the highest significance in most of our studied quantum projects.

6. Discussion and implication

In this section, we further elaborate on the results shown in Section 5 and highlight the implications of the results to the researchers and the developers of quantum software.

As the initial step, we investigated the representation of maintainability and reliability in the overall technical debts in quantum software systems as shown in Fig. 6. We then broke down the technical debt based on the severity (in Fig. 7) and examine the rules distributions across the different categories of technical debt. We found that about 80% of the technical debts are related to code smells and most of these debts are related to critical and major severity types. We also highlighted some of the examples for critical debts as reported in Table 3. We believe that this information can help practitioners prioritize resource allocation to refactor their code, specifically on the code with critical technical debts.

In our manual investigation of the quantum project of compiler category, we observed that technical debts are found most in source code for handling and generating graphs, benchmarks, unreachable code (due to inappropriate exit points, control flow or jump statements), and unexpected expressions. We also observed code smells due to multiple empty blocks of codes detected as 'suspicious' in the source code generating Python list to store gates which is of Hermitian conjugate, in the Assembly project. The related source code implementing the in-memory storage of circuits were also associated with other technical debts such as design, unused code, brain-overload (related to cognitive complexity). Implementing in-memory storage of circuits can be helpful in quantum Assembly for easy random access to any gate of the circuit, for instance, during circuit optimizations. These features are particular in quantum software. Usually, a typical quantum program consists of blocks of code, each of which contains both classical and quantum components to execute classical instructions and quantum instruction, respectively. Classical instructions operate on the state of classical bits and apply conditional statements. They are also used for post processing the outcome of measurements on qubits, while quantum instructions operate on the state of qubits and measure the qubit values (Cross et al., 2017). Studies on traditional software by Marcilio et al. (2019) revealed that the problems regarding packages and exception handling contain the biggest proportion of technical debts in their studied traditional projects. Digkas et al. (2018) also reported that the most frequent technical debts are related to resource management, null pointer, and exception handling problems.

In Table 4 we showed that a few types of technical debts contribute to about 80% and more of the overall technical debts. These dominating types include 'code convention', 'unused', 'design issues' (the design of the code is questionable, *e.g.,* duplicate string literals), 'brain-overload', 'obsolete', 'accessibility,' 'cwe,' 'confusing' and 'error-handling'. Our finding is in line with the Pareto principle hence its application. We have also shown that the type 'code convention' dominates in more than 50% of the studied quantum categories. Therefore, we recommend that quantum developers should follow the coding convention to avoid confusion and allow efficient team collaboration.

Lenarduzzi et al. (2019) have shown that constraints such as time and budgets are the root causes of technical debts. Also, factors such as refactoring efforts and architectural changes have been mentioned to influence technical debt in the software projects as compared to the changes in the line of code (Martini et al., 2015b; Molnar and Motogna, 2017). We investigated how and when the technical debts are introduced into the code base of quantum software systems to improve our understanding and verify whether similar trends can be identified for technical debt in a quantum software system. Fig. 9 shows how the code changes are related to technical debts and how these trends evolve over time. We found that most of the technical debts are associated with the initial versions, where most new code or source files are added. Our result indicates that less than 20% of the studied files with large file size introduced most of the technical debts across the studied snapshots of the quantum projects. This result is inline with the studies on traditional software (Molnar and Motogna, 2017; Molnar et al., 2019; Abidi et al., 2021). Our finding suggests that quantum software developers should pay attention to the code quality and code size, especially when new files are added. They should more carefully review and test the commits with a large number of changes. Quantum developers should also consider using the traditional static code analysis tools to detect and monitor their source code at the early stage of development to reduce future maintenance costs. In the future, we plan to examine the lifespan of quantum software faults and how developers prioritize resolving these faults based on the types and severity.

In Table 6, we found that there is a statistically significant correlation between the technical debts and the number of fault-inducing commits in the studied quantum projects. We observed that the debt types related to cognitive complexity (brain-overload), code design, redundant code and error-handling show a significant correlation with fault-inducing commit occurrences in about 60% of the quantum categories. We recommend that developers should use these metrics as measures to predict faults in quantum software systems especially when the developers are working on projects related to 'Full-stack Library', 'Quantum-annealing', 'Quantum-chemistry', 'Quantum-algorithms', and 'Compiler'. Also, as we observed the statistical significant correlation between design-related technical and faults introduced in most (60%) of the studied quantum categories, we recommend that practitioners should implement more systematic testing on quantum software systems that reflect the structural nature of the quantum software. We suggest that the quality assurance team for quantum software systems should effectively allocate more resources and time to thoroughly validate and test the complex components and the components related to error handling before integrating new commits.

We suggest that developers should use existing static analysis tools to detect potential problems that also happen in traditional software. We also appeal the quantum software community to introduce new tools to detect quantum-specific errors or technical debts.

7. Threats to validity

There are several threats that can potentially affect the validity of our study. In this section, we discuss the threats to validity of our study by following the guidelines for case study research. We followed the best practice defined to evaluate our study as presented by Runeson and Höst (2009).

Threats to construct validity. are concerned with the relationship between theory and observation. We used the SZZ algorithm to identify the changes that introduced faults. This algorithm assumes that faults are introduced by the lines that are later fixed by fault-fixing commits. However, in a fault-fixing commit, not all of the changed lines are used to fix faults. To mitigate this threat, we used the PyDriller which eliminate the candidates of fault-inducing commits that only changed white spaces. Muse et al. (2020) manually validated the precision of PyDriller and showed that the tool only yielded 6% of false positives. In addition, we used the topic modeling technique to automatically extract the characteristics of the fault-inducing changes. Some characteristics may be missed during this automatic process. In future work, we plan to select a sample of the fault-inducing commits, and manually analyze the root causes and the symptoms of the faults.

Threats to internal validity. are concerned with the factors that may affect a dependent variable and were not considered in the study. We manually selected quantum related projects and grouped the projects into different categories. To minimize this threat, the project selection and categorization were performed by two of the authors independently. They discussed each of the discrepancies until a mutual agreement was reached. We shared our dataset online.[1] Future studies can replicate and validate our results.

Threats to conclusion validity. are concerned with the relationship between the treatment and the outcome. We detect the technical debts from quantum software by using a static analysis tool (SonarQube), which is designed for traditional software. However, to mitigate this threat we limit our studied projects to be those that are written in traditional programming languages and can be analyzed by traditional static code analysis tools. Validating our results on the software running specifically on quantum computers will be our future work. To validate the detection accuracy of SonarQube on technical debts, we manually examined 500 technical debts identified by the tool and observed that 98% of the technical debts are correct. Throughout our discussion in Section 5, we highlighted the technical debts that are validated in our manual validation. In addition, we relied on a set of rules defined by SonarQube to identify different types of technical debts. These rules may not capture all possible technical debts. In the future, we plan to use other equivalent tools to verify our results and perform manual analyses to find out which kinds of debts cannot be identified by the state-of-the-art tools and will suggest the software engineering community to improve the static analysis tools on this aspect.

Threats to external validity. are concerned with the generalizability of our results. To study the characteristics of technical debts and faults in quantum software, we analyzed 118 open-source quantum projects from GitHub. Our selected projects are related to different domains of quantum computing. Our results can be considered as a reference for quantum developers and researchers to improve their software quality in terms of maintainability and reliability. However, our results may not generalize to all quantum software. Future studies are welcome to replicate and validate our work in other quantum projects.

8. Conclusion

As for traditional software, the maintenance of quantum software systems is equally important because the maintenance practices can affect the quality of a whole quantum computing system. In this study, we selected 118 open-source quantum computing related projects from GitHub. We empirically studied the

[1] https://github.com/openjamoses/JSS-Replication.

Table A.8
The names of GitHub repositories of the studied Quantum projects in each category.

	Category	The studied repositories (separated by comma)	Total
1	Full-stack Library or Framework (Fstack)	Qiskit/qiskit-terra, microsoft/Quantum, microsoft/QuantumLibraries, microsoft/Quantum-NC, rigetti/pyquil, Qiskit/qiskit, Qiskit/qiskit-ibmq-provider, qiskit-community/qiskit-js, qiskit-community/qiskit-vscode, rigetti/rpcq, ProjectQ-Framework/ProjectQ, QuantumPackage/qp2, XanaduAI/strawberryfields, Blueqat/Blueqat, quantumlib/Cirq, softwareQinc/staq	16
2	Experimentation (Exp)	m-labs/artiq, sedabull/quantum-shell, qutech/qupulse, iitis/QuantumInformation.jl, lneuhaus/pyrpl, BBN-Q/Qlab, BBN-Q/PyQLab	8
3	Simulator (Sim)	softwareQinc/qpp, QuantumBFS/YaoBlocks.jl, issp-center-dev/HPhi, SoftwareQuTech/SimulaQron, QuantumBFS/Yao.jl, Strilanc/Quirk, Qiskit/qiskit-aer, qutip/qutip, QuEST-Kit/QuEST, vm6502q/qrack, qulacs/qulacs, quantumlib/OpenFermion-Cirq, microsoft/qmt, Approximates/dotBloch, Qiskit/qiskit-jku-provider, ngnrsaa/qflex, aparent/QCViewer, rigetti/qvm, marvel-nccr/quantum-mobile	20
4	Cryptography (Crypt)	theQRL/QRL, exaexa/codecrypt, BBN-Q/QGL, mupq/pqm4	4
5	Compiler (Comp)	microsoft/qsharp-compiler, QE-Lab/OpenQL, Quantomatic/pyzx	3
6	ToolKit (Tkit)	evaleev/libint, qojulia/QuantumOptics.jl, jcmgray/quimb, XanaduAI/pennylane, OriginQ/QPanda-2, CQCL/pytket, bloomberg/quantum, rigetti/forest-benchmarking, QInfer/python-qinfer, zoran-cuckovic/QGIS-visibility-analysis, qubekit/QUBEKit, tensorflow/quantum, redhat-cip/openstack-quantum-puppet, boschmitt/tweedledum, TRIQS/triqs, QuTech-Delft/qtt, lmacken/quantumrandom, qucat/qucat, Qiskit/qiskit-ignis, orbkit/orbkit,deepchem/deepchem, aiidateam/aiida-quantumespresso	22
7	Algorithms (Algo)	qrefine/qrefine, mabuchilab/QNET, Qiskit/qiskit-aqua, quantumlib/OpenFermion, rigetti/grove, netket/netket, XanaduAI/thewalrus, qucontrol/krotov, aeantipov/pomerol, Q-solvers/EDLib, dwave-examples/factoring, PanPalitta/phase_estimation, ProjectQ-Framework/FermiLib, JoshuaSBrown/QC_Tools	14
8	Annealing (Ann)	shinmorino/sqaod, dwavesystems/qbsolv, dwavesystems/dimod, dwavesystems/dwave-system, dwavesystems/dwavebinarycsp, dwavesystems/penaltymodel	6
9	Chemistry (Chem)	ValeevGroup/mpqc, pyscf/pyscf, MolSSI-BSE/basis_set_exchange, MolSSI/QCElemental, MolSSI/QCEngine, MolSSI/QCFractal, SebWouters/CheMPS2, Quantum-Dynamics-Hub/libra-code, QMCPACK/qmcpack, cp2k/cp2k, votca/xtp, hande-qmc/hande, tmancal74/quantarhei, LCPQ/quantum_package, vonDonnerstein/QuantumLab.jl, ericchansen/q2mm, aoterodelaroza/critic2, GQCG/GQCP, qcdb/qcdb	19
10	Assembly (Asm)	valeevGroup/mpqc, MolSSI/QCElemental, MolSSI/QCEngine, QMCPACK/qmcpack, SebWouters/CheMPS2	5

distribution and evolution of technical debts in these projects and the relationship between technical debts and fault occurrences. We observed that a few types of technical debts (related to code convention violation, error-handling, and code complexity) dominate the total number of technical debts detected from our studied projects. There is a strong correlation between technical debts and post-release faults. Particularly, files with high code complexity are more likely to lead to faults. We also found that code changes related to configuration files and dependency management tend to be highly fault-prone, which need more attention when developers perform code review and testing. Based on the findings reported in this paper, we formulate the following recommendations:

- For the technical debt detection, we recommend that quantum software developers use the existing static analysis tools to examine their code.
- We also recommend that practitioners prioritize resource allocation to refactor their source code, especially the code with critical technical debts. Also, quantum developers should follow the coding convention to avoid confusion, especially to support team collaboration.
- Quantum developers should pay attention to the code quality and code size, especially when new files are added. They should review and test carefully the commits with a large number of changes.
- Code reviewers and quantum quality assurance team should use metrics like code convention, code redundancy, error-handling, and the cognitive complexity of the code to predict faulty commits.
- New tools should be introduced to support identifying quantum-specific problems, such as the technical debts and faults that only occur in a quantum software system.
- Future works are appealed to study other aspects of quantum software in terms of maintenance and reliability, such as code review, verification methods to ensure the correctness of a quantum program, and practical fault detection techniques for supporting quantum systematic testing and debugging.

CRediT authorship contribution statement

Moses Openja: Conceptualization, Methodology, Visualization, Writing, Data curation. **Mohammad Mehdi Morovati:** Conceptualization, Methodology, Data curation, Writing – original draft. **Le An:** Conceptualization, Methodology, Writing – original draft, Investigation. **Foutse Khomh:** Conceptualization, Methodology, Supervision, Reviewing and editing. **Mouna Abidi:** Conceptualization, Methodology, Writing – reviewing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

We thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for funding this project.

Appendix A. Sample quantum projects in each categories

The GitHub repositories are shown as: `<owner_name>/<repository_name>` (see Table A.8).

Appendix B. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jss.2022.111458.

References

Abhari, A.J., Faruque, A., Dousti, M.J., Svec, L., Catu, O., Chakrabati, A., Chiang, C.-F., Vanderwilt, S., Black, J., Chong, F., 2012. Scaffold: Quantum Programming Language. Tech. rep., Princeton Univ NJ Dept of Computer Science.

Abidi, M., Rahman, M.S., Openja, M., Khomh, F., 2021. Are multi-language design smells fault-prone? an empirical study. ACM Trans. Softw. Eng. Methodol. (TOSEM) 30 (3), 1–56.

Asaduzzaman, M., Bullock, M.C., Roy, C.K., Schneider, K.A., 2012. Bug introducing changes: A case study with Android. In: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR). pp. 116–119. http://dx.doi.org/10.1109/MSR.2012.6224267.

Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C., 2016. Managing technical debt in software engineering (dagstuhl seminar 16162). In: Dagstuhl Reports, Vol. 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Avgeriou, P.C., Taibi, D., Ampatzoglou, A., Fontana, F.A., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, N., Pigazzini, I., et al., 2020. An overview and comparison of technical debt measurement tools. IEEE Softw..

Banker, R.D., Davis, G.B., Slaughter, S.A., 1998. Software development practices, software complexity, and software maintenance performance: A field study. Manage. Sci. 44 (4), 433–450.

Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., Strollo, O., 2012. When does a refactoring induce bugs? An empirical study. In: Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation. In: SCAM '12, IEEE Computer Society, USA, pp. 104–113. http://dx.doi.org/10.1109/SCAM.2012.20.

Bernardi, M.L., Canfora, G., Di Lucca, G.A., Di Penta, M., Distante, D., 2012. Do developers introduce bugs when they do not communicate? The case of eclipse and mozilla. In: 2012 16th European Conference on Software Maintenance and Reengineering. pp. 139–148. http://dx.doi.org/10.1109/CSMR.2012.24.

Besker, T., Martini, A., Bosch, J., 2018. Technical debt cripples software developer productivity: a longitudinal study on developers' daily software development work. In: Proceedings of the 2018 International Conference on Technical Debt. pp. 105–114.

Besker, T., Martini, A., Bosch, J., 2019. Technical debt triage in backlog management. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). IEEE, pp. 13–22.

Bloch, A., 2003. Murphy's Law. Penguin.

Borg, M., Svensson, O., Berg, K., Hansson, D., 2019. SZZ unleashed: An open implementation of the SZZ algorithm - Featuring example usage in a study of just-in-time bug prediction for the jenkins project. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation. In: MaLTeSQuE 2019, Association for Computing Machinery, New York, NY, USA, pp. 7–12. http://dx.doi.org/10.1145/3340482.3342742.

Brown, W.H., Malveau, R.C., McCormick, H.W., Mowbray, T.J., 1998. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, Inc..

Businge, J., Openja, M., Kavaler, D., Bainomugisha, E., Khomh, F., Filkov, V., 2019. Studying android app popularity by cross-linking GitHub and google play store. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 287–297. http://dx.doi.org/10.1109/SANER.2019.8667998.

Businge, J., Openja, M., Nadi, S., Bainomugisha, E., Berger, T., 2018. Clone-based variability management in the android ecosystem. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 625–634. http://dx.doi.org/10.1109/ICSME.2018.00072.

Businge, J., Openja, M., Nadi, S., Berger, T., 2022. Reuse and maintenance practices among divergent forks in three software ecosystems. Empir. Softw. Eng. 27 (2), 1–47.

Cairo, A.S., Carneiro, G.d.F., Monteiro, M.P., 2018. The impact of code smells on software bugs: A systematic literature review. Information 9 (11), 273.

Campbell, G.A., 2018. Cognitive Complexity-A New Way of Measuring Understandability. SonarSource SA.

Canfora, G., Ceccarelli, M., Cerulo, L., Di Penta, M., 2011. How long does a bug survive? An empirical study. In: 2011 18th Working Conference on Reverse Engineering. pp. 191–200. http://dx.doi.org/10.1109/WCRE.2011.31.

Cataldo, M., Herbsleb, J.D., Carley, K.M., 2008. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 2–11.

Cook, S.A., Mitchell, D.G., 1997. Finding hard instances of the satisfiability problem: A survey. Satisf. Probl.: Theory Appl. 35, 1–17.

Cross, A.W., Bishop, L.S., Smolin, J.A., Gambetta, J.M., 2017. Open quantum assembly language. arXiv:1707.03429.

Čubranić, D., Murphy, G.C., 2003. Hipikat: Recommending pertinent software development artifacts. In: Proceedings of the 25th International Conference on Software Engineering. In: ICSE '03, IEEE Computer Society, USA, pp. 408–418.

Cunningham, W., 1992a. The WyCash portfolio management system. In: Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum). In: OOPSLA '92, Association for Computing Machinery, New York, NY, USA, pp. 29–30. http://dx.doi.org/10.1145/157709.157715.

Cunningham, W., 1992b. The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger 4 (2), 29–30.

2021. Common Weakness Enumeration (CWE):a community-developed list of software and hardware weakness types. URL http://cwe.mitre.org/.

Dale, M.R., Izurieta, C., 2014. Impacts of design pattern decay on system quality. In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 1–4.

Deutsch, D., 1985. Quantum theory, the Church–Turing principle and the universal quantum computer. Proc. R. Soc. Lond. Ser. A Math. Phys. Eng. Sci. 400 (1818), 97–117.

Digkas, G., Lungu, M., Avgeriou, P., Chatzigeorgiou, A., Ampatzoglou, A., 2018. How do developers fix issues and pay back technical debt in the apache ecosystem? In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 153–163.

Digkas, G., Lungu, M., Chatzigeorgiou, A., Avgeriou, P., 2017. The evolution of technical debt in the apache ecosystem. In: European Conference on Software Architecture. Springer, pp. 51–66.

Dunford, R., Su, Q., Tamang, E., 2014. The pareto principle.

Ekert, A.K., 1991. Quantum cryptography based on Bell's theorem. Phys. Rev. Lett. 67 (6), 661.

Ell, J., 2013. Identifying failure inducing developer pairs within developer networks. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 1471–1473. http://dx.doi.org/10.1109/ICSE.2013.6606748.

Eyolfson, J., Tan, L., Lam, P., 2011. Do time of day and developer experience affect commit bugginess? In: Proceedings of the 8th Working Conference on Mining Software Repositories. In: MSR '11, Association for Computing Machinery, New York, NY, USA, pp. 153–162. http://dx.doi.org/10.1145/1985441.1985464.

Fingerhuth, M., Babej, T., Wittek, P., 2018. Open source software in quantum computing. PLoS One 13 (12), e0208561.

Finnila, A.B., Gomez, M., Sebenik, C., Stenson, C., Doll, J.D., 1994. Quantum annealing: a new method for minimizing multidimensional functions. Chem. Phys. Lett. 219 (5–6), 343–348.

Fischer, M., Pinzger, M., Gall, H., 2003. Populating a release history database from version control and bug tracking systems. In: International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.. pp. 23–32. http://dx.doi.org/10.1109/ICSM.2003.1235403.

Fowler, M., Beck, K., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

Fox, J., Weisberg, S., 2011. Multivariate linear models in r. In: An R Companion to Applied Regression. Los Angeles: Thousand Oaks.

Garhwal, S., Ghorani, M., Ahmad, A., 2019. Quantum programming language: A systematic review of research topic and top cited languages. Arch. Comput. Methods Eng. 1–22.

GitHub, GitHub issues. URL https://docs.github.com/en/free-pro-team@latest/github/managing-your-work-on-github/about-issues.

GitHub, 2021. Github rest API. URL https://developer.github.com/v3/.

Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B., 2013. Quipper: a scalable quantum programming language. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 333–342.

Gyimothy, T., Ferenc, R., Siket, I., 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans. Softw. Eng. 31 (10), 897–910.

Hastie, T., Tibshirani, R., Friedman, J., 2009. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Science & Business Media.

Herbold, S., Trautsch, A., Trautsch, F., Ledel, B., 2022. Problems with SZZ and features: An empirical study of the state of practice of defect prediction data collection. Empir. Softw. Eng. 27 (2), 1–49.

IEEE Standards Coordinating Committee, et al., 1990. IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990), Vol. 169. IEEE Computer Society, Los Alamitos, CA.

Ivankova, N.V., Creswell, J.W., Stick, S.L., 2006. Using mixed-methods sequential explanatory design: From theory to practice. Field Methods 18 (1), 3–20.

JavadiAbhari, A., Patil, S., Kudrow, D., Heckey, J., Lvov, A., Chong, F.T., Martonosi, M., 2015. ScaffCC: Scalable compilation and analysis of quantum programs. Parallel Comput. 45, 2–17.

Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2013. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. 39 (6), 757–773. http://dx.doi.org/10.1109/TSE.2012.70.

Kaye, P., Laflamme, R., Mosca, M., et al., 2007. An Introduction to Quantum Computing. Oxford University Press.

Khomh, F., Gueheneuc, Y.-G., Antoniol, G., 2009. Playing roles in design patterns: An empirical descriptive and analytic study. In: 2009 IEEE International Conference on Software Maintenance. pp. 83–92. http://dx.doi.org/10.1109/ICSM.2009.5306327.

Kiefl, N., Hagel, G., 2020. Software engineering education of classical computing vs. quantum computing: A competency-centric approach. In: Proceedings of the 4th European Conference on Software Engineering Education. In: ECSEE '20, Association for Computing Machinery, New York, NY, USA, pp. 27–31. http://dx.doi.org/10.1145/3396802.3396816.

Kim, S., Whitehead, E.J., 2006. How long did it take to fix bugs? In: Proceedings of the 2006 International Workshop on Mining Software Repositories. In: MSR '06, Association for Computing Machinery, New York, NY, USA, pp. 173–174. http://dx.doi.org/10.1145/1137983.1138027.

Kim, S., Whitehead, E.J., Zhang, Y., 2008. Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng. 34 (2), 181–196. http://dx.doi.org/10.1109/TSE.2007.70773.

Kim, S., Zimmermann, T., Pan, K., James Jr., E., et al., 2006. Automatic identification of bug-introducing changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06). IEEE, pp. 81–90.

Kim, S., Zimmermann, T., Whitehead Jr., E.J., Zeller, A., 2007. Predicting faults from cached history. In: 29th International Conference on Software Engineering (ICSE'07). pp. 489–498. http://dx.doi.org/10.1109/ICSE.2007.66.

Knill, E., 1996. Conventions for Quantum Pseudocode. Tech. rep., Los Alamos National Lab., NM (United States).

Knill, E., Nielsen, M., 2000. Encyclopedia of Mathematics, Supplement III, Chapter Theory of Quantum Computation. Kluwer Academic Publishers.

Ko, A.J., Myers, B.A., Coblenz, M.J., Aung, H.H., 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans. Softw. Eng. 32 (12), 971–987. http://dx.doi.org/10.1109/TSE.2006.116.

Kruchten, P., Nord, R.L., Ozkaya, I., 2012. Technical debt: From metaphor to theory and practice. Ieee Softw. 29 (6), 18–21.

Krüger, T., Mauerer, W., 2020. Quantum annealing-based software components: An experimental case study with SAT solving. arXiv preprint arXiv:2005.05465.

Lenarduzzi, V., Orava, T., Saarimäki, N., Systa, K., Taibi, D., 2019. An empirical study on technical debt in a finnish SME. In: 2019 FACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). pp. 1–6.

Letouzey, J.-L., 2012. The SQALE method for evaluating technical debt. In: Proceedings of the Third International Workshop on Managing Technical Debt. In: MTD '12, IEEE Press, pp. 31–36.

Letouzey, J.-L., Coq, T., 2010. The sqale analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code. In: 2010 Second International Conference on Advances in System Testing and Validation Lifecycle. IEEE, pp. 43–48.

Letouzey, J.-L., Ilkiewicz, M., 2012. Managing technical debt with the sqale method. IEEE Softw. 29 (6), 44–51.

Li, Z., Avgeriou, P., Liang, P., 2015. A systematic mapping study on technical debt and its management. J. Syst. Softw. 101 (C), 193–220. http://dx.doi.org/10.1016/j.jss.2014.12.027.

Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W., Pinto, G., 2019. Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). IEEE, pp. 209–219.

Martin, F., 2019a. Is high quality software worth the cost?. URL https://martinfowler.com/articles/is-quality-worth-cost.html.

Martin, F., 2019b. Technical debt. URL https://martinfowler.com/bliki/TechnicalDebt.html.

Martini, A., Bosch, J., Chaudron, M., 2015a. Investigating architectural technical debt accumulation and refactoring over time. Inf. Softw. Technol. 67 (C), 237–253. http://dx.doi.org/10.1016/j.infsof.2015.07.005.

Martini, A., Bosch, J., Chaudron, M., 2015b. Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. Inf. Softw. Technol. 67, 237–253.

Mateen, A., Akbar, M.A., 2016. Estimating software reliability in maintenance phase through ann and statistics. arXiv preprint arXiv:1605.00774.

Miszczak, J.A., 2012. High-level structures for quantum computing. Synth. Lect. Quantum Comput. 4 (1), 1–129.

Mockus, Votta, 2000. Identifying reasons for software changes using historic databases. In: Proceedings 2000 International Conference on Software Maintenance. pp. 120–130. http://dx.doi.org/10.1109/ICSM.2000.883028.

Moguel, E., Berrocal, J., García-Alonso, J., Murillo, J.M., A Roadmap for Quantum Software Engineering: applying the lessons learned from the classics.

Molnar, A., Motogna, S., 2017. Discovering maintainability changes in large software systems. In: Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement. In: IWSM Mensura '17, Association for Computing Machinery, New York, NY, USA, pp. 88–93. http://dx.doi.org/10.1145/3143434.3143447.

Molnar, A.-J., Motogna, S., 2020. Long-term evaluation of technical debt in open-source software. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). pp. 1–9.

Molnar, A.-J., Neamţu, A., Motogna, S., 2019. Longitudinal evaluation of software quality metrics in open-source applications. In: Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering. In: ENASE 2019, SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT, pp. 80–91. http://dx.doi.org/10.5220/0007725600800091.

Montanaro, A., 2016. Quantum algorithms: an overview. Npj Quantum Inf. 2 (1), 1–8.

Morales, R., Soh, Z., Khomh, F., Antoniol, G., Chicano, F., 2017. On the use of developers' context for automatic refactoring of software anti-patterns. J. Syst. Softw. 128, 236–251. http://dx.doi.org/10.1016/j.jss.2016.05.042, URL https://www.sciencedirect.com/science/article/pii/S0164121216300632.

Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M., 2017. Curating github for engineered software projects. Empir. Softw. Eng. 22 (6), 3219–3253.

Muse, B.A., Rahman, M.M., Nagy, C., Cleve, A., Khomh, F., Antoniol, G., 2020. On the prevalence, impact, and evolution of SQL code smells in data-intensive systems. In: Proceedings of the 17th International Conference on Mining Software Repositories. pp. 327–338.

Nielsen, M.A., Chuang, I., 2002. Quantum Computation and Quantum Information. American Association of Physics Teachers.

Openja, M., Adams, B., Khomh, F., 2020. Analysis of modern release engineering topics : – A large-scale study using StackOverflow –. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 104–114. http://dx.doi.org/10.1109/ICSME46990.2020.00020.

Openja, M., Majidi, F., Khomh, F., Chembakottu, B., Li, H., 2022. Studying the practices of deploying machine learning projects on docker. In: The International Conference on Evaluation and Assessment in Software Engineering 2022. In: EASE 2022, Association for Computing Machinery, New York, NY, USA, pp. 190–200. http://dx.doi.org/10.1145/3530019.3530039.

Orús, R., 2014. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. Ann. Physics 349, 117–158. http://dx.doi.org/10.1016/j.aop.2014.06.013, URL http://www.sciencedirect.com/science/article/pii/S0003491614001596.

Pérez-Castillo, R., 2020. Reengineering of information systems toward classical-quantum systems. In: QANSWER. pp. 64–70.

Pérez-Delgado, C.A., Perez-Gonzalez, H.G., 2020. Towards a Quantum Software Modeling Language. In: ICSEW'20, Association for Computing Machinery, New York, NY, USA, pp. 442–444. http://dx.doi.org/10.1145/3387940.3392183.

Piattini, M., Peterssen, G., Pérez-Castillo, R., 2020a. Quantum computing: A new software engineering golden age. ACM SIGSOFT Softw. Eng. Notes 45 (3), 12–14.

Piattini, M., Peterssen, G., Pérez-Castillo, R., Hevia, J.L., Serrano, M.A., Hernández, G., de Guzmán, I.G.R., Paradela, C.A., Polo, M., Murina, E., et al., 2020b. The talavera manifesto for quantum software engineering and programming. In: QANSWER. pp. 1–5.

Pigoski, T.M., 1996. Practical Software Maintenance: Best Practices for Managing Your Software Investment. Wiley Publishing.

2021. Quantum open source foundation (QOSF). URL https://qosf.org.

Rahman, F., Devanbu, P., 2011. Ownership, experience and defects: a fine-grained study of authorship. In: 2011 33rd International Conference on Software Engineering (ICSE). pp. 491–500. http://dx.doi.org/10.1145/1985793.1985860.

Raymond, E., 1999. The cathedral and the bazaar. Knowl. Technol. Policy 12 (3), 23–49.

Reimanis, D., Izurieta, C., 2016. Towards assessing the technical debt of undesired software behaviors in design patterns. In: 2016 IEEE 8th International Workshop on Managing Technical Debt (MTD). pp. 24–27.

Rosen, C., Grawi, B., Shihab, E., 2015. Commit guru: Analytics and risk prediction of software commits. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA, pp. 966–969. http://dx.doi.org/10.1145/2786805.2803183.

Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. 14 (2), 131–164. http://dx.doi.org/10.1007/s10664-008-9102-8.

Saarimäki, N., Lenarduzzi, V., Taibi, D., 2019. On the diffuseness of code technical debt in Java projects of the apache ecosystem. In: Proceedings of the Second International Conference on Technical Debt. IEEE Press, pp. 98–107.

Saika, T., Choi, E., Yoshida, N., Haruna, S., Inoue, K., 2016. Do developers focus on severe code smells? In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 4. IEEE, pp. 1–3.

Samadhiya, D., Wang, S.-H., Chen, D., 2010. Quality models: Role and value in software engineering. In: 2010 2nd International Conference on Software Technology and Engineering, Vol. 1. pp. V1–320–V1–324. http://dx.doi.org/10.1109/ICSTE.2010.5608852.

Seacord, R.C., Plakosh, D., Lewis, G.A., 2003. Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices. Addison-Wesley Professional.

Shaydulin, R., Thomas, C., Rodeghero, P., 2020. Making quantum computing open: Lessons from open source projects. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. pp. 451–455.

Shihab, E., Jiang, Z.M., Ibrahim, W.M., Adams, B., Hassan, A.E., 2010. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 1–10.

Shor, P.W., 1999. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Rev. 41 (2), 303–332.

Siavvas, M., Tsoukalas, D., Jankovic, M., Kehagias, D., Tzovaras, D., 2020. Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises. In: Enterprise Information Systems. Taylor & Francis, pp. 1–43.

Śliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? ACM Sigsoft Softw. Eng. Notes 30 (4), 1–5.

Smit, M., Gergel, B., Hoover, H.J., Stroulia, E., 2011. Maintainability and Source Code Conventions: An Analysis of Open Source Projects, Vol. 6. Tech. Rep. TR11, University of Alberta, Department of Computing Science.

2021. SonaQube official website. URL https://www.sonarqube.org/.

Sonarqube, Sonarqube, doc 8.6 issues. URL https://docs.sonarqube.org/latest/user-guide/issues/.

2021a. SonarQube built-in rule tags. URL https://docs.sonarqube.org/latest/user-guide/built-in-rule-tags/.

2021b. SonarQube documentation rules. URL https://docs.sonarqube.org/latest/user-guide/rules/.

Spadini, D., Aniche, M., Bacchelli, A., 2018. PyDriller: Python framework for mining software repositories. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. In: ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, pp. 908–911. http://dx.doi.org/10.1145/3236024.3264598.

Spector, L., Barnum, H., Bernstein, H.J., Swamy, N., 1999. Quantum computing applications of genetic programming. Adv. Genet. Program. 3, 135–160.

Svore, K., Geller, A., Troyer, M., Azariah, J., Granade, C., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., Roetteler, M., 2018. Q# Enabling scalable quantum computing and development with a high-level DSL. In: Proceedings of the Real World Domain Specific Languages Workshop 2018. pp. 1–10.

Taba, S.E.S., Khomh, F., Zou, Y., Hassan, A.E., Nagappan, M., 2013. Predicting bugs using antipatterns. In: 2013 IEEE International Conference on Software Maintenance. IEEE, pp. 270–279.

Tan, J., Feitosa, D., Avgeriou, P., Lungu, M., 2020. Evolution of technical debt remediation in Python: A case study on the apache software ecosystem. J. Softw.: Evol. Process e2319.

Techopedia, Technical debt. URL http://www-cs-faculty.stanford.edu/~uno/abcde.html.

Thompson, J., Modi, K., Vedral, V., Gu, M., 2018. Quantum plug n'play: modular computation in the quantum regime. New J. Phys. 20 (1), 013004.

Tsoukalas, D., Kehagias, D., Siavvas, M., Chatzigeorgiou, A., 2020. Technical debt forecasting: An empirical study on open-source repositories. J. Syst. Softw. 170, 110777.

Tufano, M., Bavota, G., Poshyvanyk, D., Di Penta, M., Oliveto, R., De Lucia, A., 2017. An empirical study on developer-related factors characterizing fix-inducing commits. J. Softw.: Evol. Process 29 (1), e1797.

Ubayawardana, G.M., Karunaratna, D.D., 2018. Bug prediction model using code smells. In: 2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer). IEEE, pp. 70–77.

undefinedliwerski, J., Zimmermann, T., Zeller, A., 2005. HATARI: Raising risk awareness. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: ESEC/FSE-13, Association for Computing Machinery, New York, NY, USA, pp. 107–110. http://dx.doi.org/10.1145/1081706.1081725.

VanDoren, E., 1997. Maintenance of Operational Systems–An Overview, Software Technology Roadmap. Carnegie Mellon Software Engineering Institute.

Walkinshaw, N., Minku, L., 2018. Are 20% of files responsible for 80% of defects? In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 1–10.

Wen, M., Wu, R., Cheung, S., 2016. Locus: Locating bugs from software changes. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 262–273.

Wen, M., Wu, R., Liu, Y., Tian, Y., Xie, X., Cheung, S.-C., Su, Z., 2019. Exploring and Exploiting the Correlations between Bug-Inducing and Bug-Fixing Commits. In: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 326–337. http://dx.doi.org/10.1145/3338906.3338962.

Wikipedia, F-test. URL https://en.wikipedia.org/wiki/F-test.

Wu, R., Wen, M., Cheung, S.-C., Zhang, H., 2018. ChangeLocator: Locate crash-inducing changes based on crash reports. Empir. Softw. Engg. 23 (5), 2866–2900, URL https://doi.org/10.1007/s10664-017-9567-4.

Xiong, C.J., Li, Y.F., Xie, M., Ng, S.H., Goh, T.N., 2009. A model of open source software maintenance activities. In: 2009 IEEE International Conference on Industrial Engineering and Engineering Management. pp. 267–271.

Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., Bairavasundaram, L., 2011. How do fixes become bugs? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 26–36.

Ying, M., 2016. Foundations of Quantum Programming. Morgan Kaufmann.

Zar, J.H., 2005. Spearman rank correlation. In: Encyclopedia of Biostatistics, Vol. 7. Wiley Online Library.

Zhao, J., 2020. Quantum software engineering: Landscapes and horizons. arXiv preprint arXiv:2007.07047.

Zhong, H., Su, Z., 2015. An empirical study on real bug fixes. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1. In: ICSE '15, IEEE Press, pp. 913–923.