2-2021

An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks

Jiakun LIU

Qiao HUANG

Xin XIA

Emad SHIHAB

David LO
Singapore Management University, davidlo@smu.edu.sg

See next page for additional authors

Empirical Software Engineering (2021) 26:16
https://doi.org/10.1007/s10664-020-09917-5

An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks

Jiakun Liu[1] · Qiao Huang[1] · Xin Xia[2] · Emad Shihab[3] · David Lo[4] · Shanping Li[1]

Accepted: 2 October 2020Published online: 15 February 2021

Abstract
To complete tasks faster, developers often have to sacrifice the quality of the software. Such compromised practice results in the increasing burden to developers in future development. The metaphor, technical debt, describes such practice. Prior research has illustrated the negative impact of technical debt, and many researchers investigated how developers deal with a certain type of technical debt. However, few studies focused on the removal of different types of technical debt in practice. To fill this gap, we use the introduction and removal of different types of self-admitted technical debt (i.e., SATD) in 7 deep learning frameworks as an example. This is because deep learning frameworks are some of the most important software systems today due to their prevalent use in life-impacting deep learning applications. Moreover, the field of the development of different deep learning frameworks is the same, which enables us to find common behaviors on the removal of different types of technical debt across projects. By mining the file history of these frameworks, we find that design debt is introduced the most along the development process. As for the removal of technical debt, we find that requirement debt is removed the most, and design debt is removed the fastest. Most of test debt, design debt, and requirement debt are removed by the developers who introduced them. Based on the introduction and removal of different types of technical debt, we discuss the evolution of the frequencies of different types of technical debt to depict the unresolved sub-optimal trade-offs or decisions that are confronted by developers along the development process. We also discuss the removal patterns of different types of technical debt, highlight future research directions, and provide recommendations for practitioners.

Keywords Self-admitted technical debt · Deep learning · Categorization · Empirical study

Communicated by: Gabriele Bavota

✉ Xin Xia
xin.xia@monash.edu

Extended author information available on the last page of the article.

1 Introduction

During the process of software development, developers are expected to continuously deliver high-quality products or services. However, because of time pressure, market competition, and cost reduction (Lim et al. 2012), developers are often confronted with a dilemma: a shorter completion time or better software quality. The compromised decision leads to the increasing burden in the future development life cycle. The metaphor, technical debt, first proposed by Cunningham (1993), describes such the decision.

Previous research finds that technical debt is detrimental, e.g., increasing the cost, and negatively impacting the product quality (Wehaibi et al. 2016; Zazworka et al. 2011; Fontana et al. 2012). Therefore, many researchers investigate the types of technical debt (Alves et al. 2014; Zazworka et al. 2013; Li et al. 2015) and inspect how developers deal with technical debt (Zazworka et al. 2011; Ernst et al. 2015; Klinger et al. 2011; Spínola et al. 2013). However, previous research only focused on certain types of technical debt, e.g., observing the introduction and removal of code smell to track how developers deal with design debt (Zazworka et al. 2011). One of the reasons is that observing technical debt is difficult and requires a thorough analysis of the whole project as the technical debt is often not directly visible. None of them characterize the removal of different types of technical debt at the same time.

To fill this gap, we use the self-admitted technical debt (i.e., SATD) as an indicator of technical debt. This is because previous research finds that most of the developers do not consider technical debt as a result of sloppy programming or poor developer discipline. Instead, they consider it as a result of intentional decisions to trade off competing concerns during development (Klinger et al. 2011). More specifically, such technical debt is the comment that is **intentionally** introduced by developers to alert the inadequacy of the solution (Potdar and Shihab 2014) and is acknowledged by developers. For example, in the open-source project TensorFlow, a comment saying *TODO(b/26910386): Identify why this infrequently causes timeouts.*, indicates that the corresponding code is problematic and needs further investigation.

In this paper, we use the introduction and removal of SATD instances in a family of software systems as an example, e.g., the development of deep learning frameworks, to find the common patterns on how developers remove different types of technical debt in this software family. We employ the development of deep learning frameworks as an example because we would like to choose a homogeneous set of projects from the same domain to minimize the domain confounding effect. The field of the development of different deep learning frameworks is the same, i.e., offering high-level programming interfaces to deep learning applications by the implementation of a range of concrete tasks, e.g., implementing core building blocks for designing, training, and validating deep neural networks. Moreover, deep learning frameworks are arguably some of the most important software systems today, due to the wide use of deep learning applications and its prevalence in health (Litjens et al. 2017), cars (Sallab et al. 2017; Huval et al. 2015; Al-Qizwini et al. 2017; Shalev-Shwartz et al. 2016), etc, i.e., life-impacting software. Hence, the effective and efficient maintainability of deep learning frameworks is of critical importance. However, the deep learning related techniques are still rapidly advancing, with many cutting edge technologies being continuously proposed, which cover a wide range of knowledge, e.g., Generative Adversarial Networks (GAN)[1] (Goodfellow et al. 2014), Tensor Processing Unit (TPU)[2]

[1] https://www.tensorflow.org/api_docs/python/tf/contrib/gan
[2] https://www.tensorflow.org/api_docs/python/tf/contrib/tpu

(Jouppi et al. 2017), and Batch Normalization[3] (Ioffe and Szegedy 2015). Developers have to implement these novel techniques in time to win the fierce market competition, which increases the risk of technical debt at the same time. As a part of the effective and efficient maintainability of deep learning frameworks, the removal of technical debt is a crucial aspect.

To do so, we first extract all the comments in all versions of files, and then we identify the SATD instances by SATD-detector (Liu et al. 2018; Huang et al. 2018). When we started our research, SATD-detector (Liu et al. 2018; Huang et al. 2018) was the most advanced NLP based algorithm to automatically identify the SATD instances. We re-train the SATD-detector with the comments that are presented in Liu et al. (2020)'s work, where they manually labeled the comments in the latest stable version of the deep learning frameworks we studied. Finally, based on Liu et al. (2020)'s work, we manually label the detected SATD instances into different types by card-sorting (Spencer 2009). We identify the 7 types of technical debts that are the same as Liu et al. (2020)'s work: design debt (i.e., sub-optimal design), documentation debt (i.e., incomplete documentation), defect debt (i.e., unresolved defects), requirement debt (i.e., incomplete implementation of the methods), test debt (i.e., deficiencies in tests), algorithm debt (i.e., sub-optimal algorithm), and compatibility debt (i.e., immature dependencies). We observe that 75 % of SATD instances that are introduced before the latest stable version are removed in 299 days at the most (for PyTorch). To avoid the bias caused by the SATD instances that are introduced recently before the latest stable version (i.e., right censoring) (Quesenberry et al. 1989), we investigate the introduction of different types of technical debt instances that are introduced over one year before the latest stable release version. Then, we characterize their removal before the latest stable release version. More specifically, with the data, we characterize the removal of technical debt by exploring several questions:

(1) **Which types of technical debt are prevalently introduced along the development process?**

The distribution of different types of technical debt that are introduced along the development process can reflect what kind of sub-optimal trade-offs or decisions are made by developers during the development process. We find that developers introduce the design debt the most during the development, followed by requirement debt and algorithm debt.

(2) **Which types of technical debt are removed the most?**

The distribution of different types of technical debt among the removed SATD instances along the development process can reflect the allocation of developers' effort in the resolution of technical debt. The proportion of introduced technical debt instances that are removed along the development process can reflect which types of technical debt are removed the most to finish the development tasks and ensure the code quality. We find that requirement debt is removed the most, followed by design debt.

(3) **Which types of technical debt are removed the fastest?**

The survival time of the technical debt, i.e., the time interval since the introduction to the removal of technical debt, can illustrate the priority of different types of technical debt when developers resolve them. We find that design debt is removed the fastest, followed by requirement debt.

(4) **Who removes different types of technical debt?**

It is expected that the technical debt is self-removed, i.e., removed by the developer who

[3] https://mxnet.incubator.apache.org/api/python/symbol/symbo-l.html#mxnet.symbol.BatchNorm

introduces it. As a result, most of test debt, design debt, and requirement debt are removed by the developers who introduced them.

Based on the introduction and removal of different types of technical debt instances, we depict the evolution of the frequencies of different types of technical debt that are presented in different stages. This evolution can illustrate the changes in developers' concerns. We also discuss the removal patterns of different types of technical debt, highlight future research directions, and provide recommendations for practitioners.

Paper Organization The remainder of this paper is structured as follows. Section 2 presents the detailed approaches we use to collect and pre-process data. Section 3 presents our research findings. Section 4 depicts the evolution of the frequencies of different types of technical debt along the development process, summarizes the removal patterns of different types of technical debt, presents the implications and actionable suggestions based on our findings, and describes some threats to the validity of this study. Section 5 presents the related work of our study, including the research works on technical debt and research works on software engineering for deep learning. We also compare the findings in our work with the findings in previous work. Finally, Section 6 concludes our study and presents future work.

2 Case Study Setup

In this section, we describe the steps that we took for project selection, project data extraction, source code comments extraction, SATD instances identification and manual classification.

2.1 Project Selection

We focus on open-source deep learning frameworks hosted in Github. We exclude deep learning applications that build upon such frameworks, or general-purpose mathematical libraries that those deep learning frameworks build upon. To do so, we first search repositories labeled by *deeplearning* and *deep learning* topics[4] in GitHub. Then, we identify the deep learning framework projects by reading the readme file of the projects. As a result, we include 7 deep learning frameworks with the largest number of stars that are written in 3 programming languages (C++, Python and Java) as subject frameworks for our study. They include: TensorFlow (shortened as TF),[5] Keras,[6] Deeplearning4j (shortened as DL4J),[7] Caffe,[8] PyTorch,[9] MXNet[10] and Microsoft Cognitive Toolkit (known as CNTK).[11]

Table 1 provides statistics of each framework in the latest stable version in our study, including the release version, the total number of lines of code, the total number of commits, the number of contributors and the main programming languages. Following the previous

[4] https://blog.github.com/2017-01-31-introducing-topics/
[5] https://github.com/tensorflow/tensorflow
[6] https://github.com/keras-team/keras
[7] https://github.com/deeplearning4j/deeplearning4j
[8] https://github.com/BVLC/caffe
[9] https://github.com/pytorch/pytorch
[10] https://github.com/apache/incubator-mxnet
[11] https://github.com/Microsoft/CNTK

Table 1 Overview of studied projects

Framework	Release	#Lines of Code	# Commits	#Contributors	Languages
TF	v1.9.0	1,821,016	34,227	1,868	Python, C++
Keras	2.2.2	42,182	4,651	782	Python
Caffe	1.0.0	76,322	4,020	318	C++
PyTorch	v0.4.0	617,255	10,835	1,010	Python, C++
MXNet	1.2.1	305,755	7,015	682	Python, C++
CNTK	v2.5.1	324,472	15,575	269	Python, C++
DL4J	0.9.1	361,366	8,375	185	Java

study by Maldonado and Shihab (2015), we calculate the total number of code lines using SLOCCount.[12]

2.2 Comment Extraction

We need the comments in all of the file history of the selected deep learning frameworks. We follow the steps performed in Maldonado et al.'s work (2017). More specifically, since we are interested in when the SATD is removed and who removed the SATD during the whole development process, we investigate the introduction and removal of SATD along with the commits on the master branch.

To do so, we first obtain all the versions of files along the master branch. We identify all Java, C++, and Python source code files currently available in the latest version of the project, and then we check out each version of the repository to get deleted files in each commit, which are currently absent in the latest stable version. We identify the rename or move of the file with Git. Finally, we obtain all versions of files by tracking all the commits done to each file.

After obtaining all versions of files in the software repositories, we discriminate between source code and comment lines. We use the srcML Toolkit,[13] which is capable of parsing source files that are coded by C++ and Java, into XML files. For Python files that are not supported by srcML, we utilize the tokenize module[14] in the Python standard library, which provides a lexical scanner for Python source code to identify all comments. We record the file name, the class name, the method name, and the comment content, as well as the meta-information of the version of the file that contains the comments, e.g., the creation date, the creation user email, the commit id, which is obtained from the version control system, i.e., Git.

To track the introduction and removal of the comment, we consider the first available file version that contains the comment as the file version that introduced the comment. Similarly, we consider the first version that the comment instance does not exist or the file where the comment exists is deleted as the removal version. The file where the comment exists being deleted also indicates that the comment does not exist. In certain cases, a comment is found in one version only (i.e., the version that it is introduced in), which indicates a scenario where the comment is introduced and removed immediately after the introduction.

[12] https://dwheeler.com/sloccount/

[13] https://www.srcml.org/

[14] https://docs.python.org/3/library/tokenize.html

Moreover, there can be inconsistent changes between the comments and the code, i.e., in some cases the comment may change but not the code, and vice versa, we will discuss this threat in Section 4.3. Finally, we extract a total of 445,149 distinct comments in all versions of the files.

2.3 Identification of SATD Instance

To identify technical debt, we follow Maldonado et al.'s work, which uses an NLP based algorithm to automatically identify the comments that indicate technical debt. When we started our research, SATD-detector (Liu et al. 2018; Huang et al. 2018) was the most advanced NLP based algorithm to automatically identify the SATD instances. More specifically, to build the model, SATD-detector preprocesses the text descriptions of comments and extracts features (i.e. words) to represent each comment at first. Then Information Gain (IG) is employed to select features that are useful for classification and remove useless features. Finally, the selected features are used to train a classifier for each project. In the prediction phase, the comment is processed to extract features. Then the features are inputted to the trained composite classifier. Finally, each sub-classifier will predict the label of the comment according to its features, and the label with the largest number of "votes" will be chosen as the final prediction result of the composite classifier.

To ensure the accuracy of SATD-detector in detecting **the SATD instances in all versions of files** in deep learning frameworks, we re-train the SATD-detector with the comments of the deep learning frameworks which are labeled in Liu et al. (2020)'s work. In Liu et al. (2020)'s work, they manually label the comments in the latest stable version of the 7 deep learning frameworks as we studied and find that there are 7,159 SATD instances. This indicates that there is an overlap between the training dataset and the test dataset. This could lead to higher performance for SATD-detector in identifying the SATD instances in all versions of files in deep learning frameworks. We report the precision and recall of the SATD-detector after labeling the SATD instances in the identified comments in Section 2.4.

2.4 Manual Classification

To determine the different SATD types, we utilize the SATD types which are found in Liu et al. (2020)'s work as a starting point, where they analyze the comments of 7 popular deep learning frameworks as we studied. They find that the technical debt in deep learning frameworks can be classified into seven categories: design debt, defect debt, documentation debt, requirement debt, test debt, algorithm debt, and compatibility debt.

In our paper, we perform two iterations of a card sorting approach (Spencer 2009) to classify 29,778 detected SATD instances in these 7 deep learning frameworks. Concretely, in the first iteration of classification, we try to ensure that our classification standard is consistent with previous work. To do so, we first randomly pick 100 comments from the dataset provided by Liu et al. (2020)'s work, then the first two authors manually classify these sentences according to Liu et al. (2020)'s work. A discussion on the disagreements with Liu et al. (2020)'s work is performed after the classification process. To validate our classification standard, the first author selected another 500 comments from the dataset provided by Liu et al. (2020)'s work and manually classified them. Then, we calculate the Cohen's kappa coefficient (McHugh 2012) and obtain a result of +0.85, which indicates a high level of agreement with the classification given by the first author and Liu et al. (2020)'s work. During this phase, the coding schema of different types of technical debt in deep learning frameworks is revised.

In the second classification iteration, the first author classifies all the 29,778 detected SATD instances. During this phase, the categories of the SATD instances in all versions of files are identified. To reduce personal bias in the manual classification of code comments, we randomly sampled a statistically representative sample of 1,000 SATD instances from the 29,778 detected SATD instances using a 95 % confidence level with a 10 % confidence interval. We invite an independent Ph.D. student, who is not an author of this paper, to manually classify the randomly sampled 1,000 SATD instances. We discuss the disagreements in Section 4.3. A high level of agreement between the classification given by the two different students is reported with Cohen's kappa coefficient of +0.79. This gives us high confidence in the dataset used in our paper.

As a result, we find that there are 24,032 SATD instances in all versions of files in deep learning frameworks. We observe that 75 % of the SATD instances that are introduced before the latest stable version are removed in 299 days at the most (for PyTorch). To avoid bias caused by the SATD instances that are introduced recently before the latest stable version (i.e., right censoring) (Quesenberry et al. 1989), we exclude the SATD instances that are introduced in one year before the latest stable release version. More specifically, we investigate the introduction of different types of technical debt instances that are introduced over one year before the latest stable release version. Then, we characterize their removal before the latest stable release version.

We discuss the performance of SATD-detector in identifying the SATD instances that are studied in our work. For 318,044 comments that are introduced over one year before the latest stable release version, 21,702 of them are identified as SATD instances by SATD-detector, and 17,576 of them are classified as SATD instances by our manual classification process. This shows that the retrained SATD-detector achieves a precision score of 0.81 as 4,126 comments are false positive (i.e., not the comments indicating technical debt). Table 3 reports the precision scores of SATD-detector in identifying the SATD instances in different projects. Besides, we observe that the false positive instances are almost uniformly introduced in different years (normalized entropy scores range from 0.76 for PyTorch to 0.97 for Keras). Therefore, moving further back into the evolutionary history of the project would not affect the performance of the SATD-detector in identifying SATD instances. To check the recall of SATD-detector in identifying the comments indicating technical debt, we randomly sampled a statistically representative sample of 100 comments from 318,044 comments using a 95 % confidence level with a 10 % confidence interval. We find that there is only 1 comment indicating technical debt. This shows that the retrained SATD-detector achieves a recall score of 0.85 as there are around 3,180 comments that are false negative (i.e., the comments indicating technical debt). We identify the following types of technical debt, which are the same as Liu et al.'s work:

(1) **Design debt** indicates sub-optimal design, e.g., misplaced code, lack of abstraction, long methods, poor implementation, workarounds, or temporary solutions on the usage of other internal functions.

Example: *"TODO(b/32239616): This kernel should be moved into Eigen and vectorized."* - [TF][15]

(2) **Defect debt** corresponds to code that behaves in unintended ways, and developers postpone repairing it because of various factors (e.g., time-consuming to resolve).

[15] tensorflow/tensorflow/core/kernels/cwise_ops.h

Example: *"Linear weights do not follow the column name. But this is a rare use case, and fixing it would add too much complexity to the code."* - [TF][16]

(3) **Documentation debt** indicates missing, inadequate or incomplete documentation that explains the corresponding part of the program.

Example: *"TODO(sibyl-vie3Poto): Write up a doc with concrete derivation and point to it from here."* - [TF][17]

(4) **Requirement debt** indicates *incompleteness* of the method, class or program at the time, which may mean that the original planned completion of the task exceeds the development schedule. It can also correspond to cases when new requirements are identified during the development of existing requirements but cannot be considered due to time pressure or other constraints.

Example: *"TODO setup for RNN"* - [DL4J][18]

(5) **Test debt** indicates the need for improvements to address deficiencies in the test suite.

Example: *"TODO(fchollet): insufficiently tested."* - [TF][19]

(6) **Compatibility debt** refers to the debt related to a project's immature dependencies on other projects, which cannot supply all qualified services, and the current implementation is a temporary workaround.

Example: *"Moved to common.cpp instead of including boost/thread.hpp to avoid a boost/NVCC issues (#1009, #1010) on OSX. Also fails on Linux with CUDA 7.0.18."* - [Caffe][20]

(7) **Algorithm debt** refers to the debt that the algorithm implemented in a deep learning framework is sub-optimal.

Example: *"TODO(Yangqing): Is there a faster way to do pooling in the channel-first case?"* - [Caffe][21]

3 Findings

In this section, we first investigate the distribution of the introduced technical debt, and then we quantify the removal of technical debt from different perspectives, such as removal rate, removal pace, and self-removal rate.

3.1 RQ1: Which Types of Technical Debt are Prevalently Introduced Along the Development Process?

Motivation In Liu et al. (2020)'s work, they observed different types of technical debt are in the latest stable version of 7 deep learning frameworks. However, it is still unclear which types of technical debt are prevalently introduced along the development process. The admitted technical debt indicates the acknowledgment of sub-optimal trade-offs or decisions during the development process. To summarize which types of sub-optimal trade-offs or

[16] tensorflow/tensorflow/python/feature_column/feature_column_test.py
[17] tensorflow/tensorflow/core/kernels/hinge-loss.h
[18] deeplearning4j/deeplearning4j-nn/src/main/java/org/deeplearning4j/nn/params/Batch-NormalizationParamInitializer.java
[19] tensorflow/tensorflow/python/keras/backend_test.py
[20] caffe/include/caffe/common.hpp
[21] caffe/src/caffe/layers/pooling_layer.cpp

decisions developers would admit more during the development process, we investigate the distribution of different types of technical debts introduced throughout the development process.

Approach To better describe the introduction of different types of technical debt along the development process, we first analyze the distribution of different types of technical debt that are introduced along the development process, then we present the distribution of different types of technical debt of all technical debt instances that are introduced one year before the latest stable version.

To characterize the distribution of different types of technical debt along the development process, we first divide the whole development process one year before the latest stable release version into ten **development phases** based on the chronological order of the commits. Then we count the number of different types of technical debt instances that are introduced in each development phase. Since different numbers of SATD instances are introduced in different development phases, we normalize different types of SATD instances by the number of total SATD instances that are introduced in that development phase. For example, in TensorFlow, there are 19,032 commits along the development process one year before the latest stable release version. We first divide the whole development process one year before the latest stable release version into 10 development phases. Each development phase has 1,903 commits. Then, we count the number of different types of SATD instances that are introduced in each development phase and normalize them with the total number of the SATD instances introduced in that phase. Figure 1 shows the distributions of different types of the introduced SATD instances along the developing process in 7 deep learning frameworks.

To check whether the difference between different types of technical debt in terms of their proportions among the SATD instances that are introduced along the development process is statistically significant, we perform a Kruskal-Wallis H test (1952). Kruskal-Wallis H test is a non-parametric test for comparing whether two or more independent samples originate from the same distribution. As a result, we find that the difference between different types of technical debt in terms of its proportions in introduced SATD instances along the development process is significant (p-value < 0.05). Then, we perform a Dunn's test with Bonferroni correction to determine which groups differ from each other group (Dunn 1961). Dunn's test can be used for the post-hoc analysis for the specific sample pairs. To calculate the effect size, we calculate the corresponding Cliff's deltas (1993). Cliff's delta is a measure of how often the values in one distribution are larger than the values in a second distribution. Table 2 presents p-values and Cliff's deltas.

Table 3 presents an overview of the distribution of different types of technical debt in different frameworks, as well as the total number of SATD instances that are introduced one year before the latest stable release version for each project. To better view the differences between different types of technical debt, we highlight the top three types in terms of the proportion in each project in bold.

Results Table 2 shows that the differences between design debt and other types of technical debt are significantly and large. Figure 1 shows that design debt is the most introduced technical debt along the development process with fluctuation in MXNet, CNTK, and DL4J. Design debt is the most introduced technical debt in most of the development phases in TensorFlow and Keras. This shows that design debt is the most common technical debt across deep learning frameworks along the development process. During the development process, developers are not satisfied with the design of the code. Developers admit the inadequacy

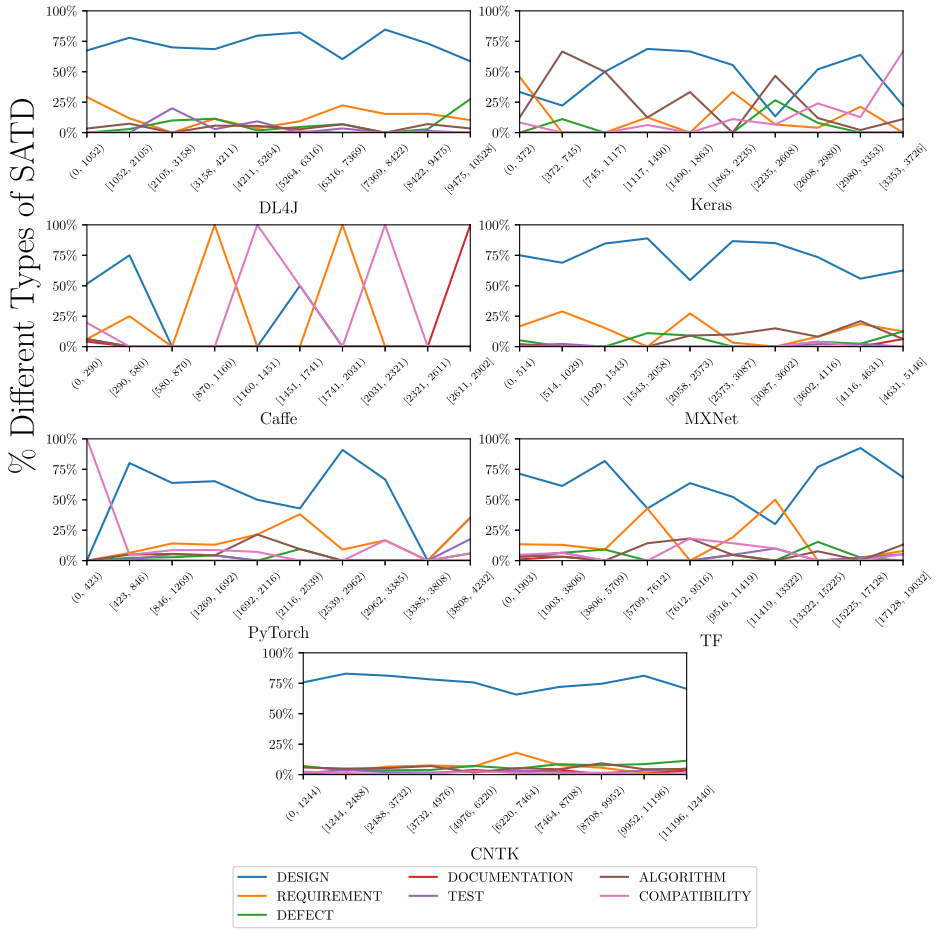

Fig. 1 Distribution of different types of SATD instances that are introduced in different development phases

of the design of the code the most. Caffe is an outlier here, design debt is the most introduced technical debt at the beginning of the development process, and following that, either requirement debt or test debt is the most introduced technical debt. One possible reason is that the Caffe planned to migrate to a new project, i.e., Caffe2. Therefore, in the later phase of development, the developers in Caffe pay attention more to the implementation of requirements and the completeness of test cases, rather than the design of code. Finally, until one year before the latest stable release version of 7 deep learning frameworks, the proportions of the introduced design debt among all introduced technical debt instances range from 76.33 % in CNTK to 48.52 % in Caffe.

Table 2 shows that the differences between requirement debt and other types of technical debt except algorithm debt are significant, and the effect sizes range from medium to large. Figure 1 shows that requirement debt is one of the second most commonly introduced technical debt along the time in 7 deep learning frameworks, e.g., in PyTorch. This shows that requirement debt is the second most common technical debt. Developers frankly wrote

Table 2 P-values and Cliff's deltas of the differences between different types of technical debt in terms of their proportions among the SATD instances that are introduced along the development process

	Algorithm	Compatibility	Defect	Design	Documentation	Requirement
Compatibility						
Defect						
Design	⋆ (+large)	⋆ (+large)	⋆ (+large)			
Documentation	⋆ (-large)	⋆ (-medium)	⋆ (-medium)	⋆ (-large)		
Requirement		⋆ (+medium)	⋆ (+medium)	⋆ (-large)	⋆ (+large)	
Test	⋆ (-medium)			⋆ (-large)		⋆ (-large)

We report the pairs of SATD types with p-value < 0.05 (i.e., significant) with ⋆ and the interpretation of corresponding Cliff's deltas

down the unaccomplished tasks in comments as a notification during the development process. In certain development phases, e.g., the 7th development phase in TensorFlow, the 6th development phase in PyTorch, requirement debt is introduced more. One possible reason is that developers are arranged to finish more requirements that exceed their ability. The unaccomplished requirements are left as requirement debt. Finally, until one year before the latest stable release version of 7 deep learning frameworks, the proportions of requirement debt instances among all introduced technical debt in different frameworks range from 7.04 % in Caffe to 16.75 % in Keras.

Table 2 shows that algorithm debt is significantly different from documentation debt and test debt, and the effect sizes range from medium to large. Figure 1 shows that different from the design debt and requirement debt that are introduced more along the development process, algorithm debt is introduced more in certain development phases. This shows that algorithm debt is the third most common technical debt. For example, in MXNet, design debt, requirement debt, and defect debt is introduced more before the 5th development phases. Since the 5th development phase, more than 10 % of the introduced technical debt is algorithm debt. One possible reason is that developers transfer their attention from the design of code and the implementation of functions to the optimization of algorithm. Finally,

Table 3 Distribution of different types of SATD that are introduced one year before the latest stable release version

Project name	TF	Keras	CNTK	Caffe	MXNet	PyTorch	DL4J	Average
Total	5,622	191	8,398	270	432	1,577	581	2,438.7
Design	**65.71 %**	**51.31 %**	**76.33 %**	**48.52 %**	**74.31 %**	**68.80 %**	**72.46 %**	**65.35 %**
Compatibility	2.81 %	12.04 %	1.70 %	**10.00 %**	1.39 %	5.64 %	0.00 %	4.80 %
Defect	3.33 %	5.24 %	**5.54 %**	4.07 %	3.01 %	3.04 %	**5.34 %**	4.22 %
Documentation	1.03 %	0.00 %	1.06 %	**20.00 %**	0.23 %	0.25 %	0.17 %	3.25 %
Test	5.12 %	0.52 %	2.66 %	2.59 %	1.16 %	2.85 %	2.58 %	2.50 %
Algorithm	**6.05 %**	**14.14 %**	5.51 %	7.41 %	**5.79 %**	**5.64 %**	4.82 %	**7.05 %**
Requirement	**15.96 %**	**16.75 %**	**7.04 %**	7.41 %	**14.12 %**	**13.76 %**	**14.63 %**	**12.81 %**
Precision	0.71	0.45	0.91	0.79	0.60	0.83	0.81	0.81

We also report the precision scores of the retrained SATD-detector in detecting to SATD instances in different deep learning frameworks

until one year before the latest stable release version of 7 deep learning frameworks, the proportions of algorithm debt in different frameworks range from 4.82 % in DL4J to 14.14 % in Keras.

Table 2 shows that documentation debt is significantly different from other types of technical debt except for test debt, and the effect size range from medium to large. Along the development process, we hard to observe the introduction of documentation debt. This shows that documentation debt is the least common debt. Considering quantities of comments and documentation of the deep learning frameworks, the documentation debt can be related to that developers seldom perform sub-optimal trade-offs or decisions related to documentation. Finally, until one year before the latest stable release version of 7 deep learning frameworks, the proportions of documentation debt in different frameworks range from 0 % in Keras to 20.0 % in Caffe.

Table 2 shows that test debt is significantly different from algorithm debt, design debt, and requirement debt, and the effect size range from medium to large. This shows that test debt is the second least introduced technical debt. The small proportions of test debt do not mean that there are fewer sub-optimal trade-offs or decisions related to the testing in the projects. One possible reason is the wide use of professional test management systems, such as QTest.[22]

Design debt is the most prevalent technical debt along the development process, followed by requirement debt and algorithm debt. Documentation debt is the least common technical debt along the development process, followed by test debt.

3.2 RQ2: Which Types of Technical Debt are Removed the Most?

Motivation In this section, we would like to characterize the removal of different types of technical debt in deep learning frameworks along the development process. In Section 3.1, we observe the prevalence of different types of technical debt in deep learning frameworks. To ensure the code quality, developers are expected to take actions (e.g., factoring) to resolve these SATD instances. However, it is still unclear which types of technical debt is removed the most along the development process.

Approach To better describe the removal of different types of technical debt along the development process, we first analyze the distribution of different types of technical debt that are removed along the development process. By doing so, we could understand which types of technical debt attract development attention more in different development phases. Then we investigate the proportion of different types of introduced SATD instances that are removed along the development process. By doing so, we could understand which types of technical debt are removed the most to finish the development tasks and ensure the code quality. Finally, we present the proportion of different types of technical debt of all technical debt instances that are removed before the latest stable version.

To describe the removal of different types of technical debt along the development process, we divide the whole development process into ten **development phases** based on the chronological order of the commits. We first analyze the distribution of different types of technical debt among the technical debt that is removed in different development phases. More specifically, we normalize the number of different types of technical debt that are

[22] https://www.qasymphony.com/software-testing-tools/qtest-manager/test-case-management/

removed in each development phase by the total number of technical debt instances that are removed in that development phase. Figure 2 shows the distribution of different types of technical debt among the removed technical debt instances in different development phases.

To check whether the differences between different types of technical debt in terms of their proportions among the removed technical debt instances along the development process are significant, we perform a Kruskal-Wallis H test (1952). Kruskal-Wallis H test is a non-parametric test for comparing whether two or more independent samples originate from the same distribution. As a result, we find that the differences between different types of technical debt in terms of their proportions in removed technical debt instances are significant (p-value < 0.05). Then, we perform a Dunn's test with Bonferroni correction to determine which groups differ from each other group (Dunn 1961). Dunn's test can be used for the post-hoc analysis for the specific sample pairs. To calculate the effect size, we calculate the corresponding Cliff's deltas (1993). Cliff's delta is a measure of how often the values in one distribution are larger than the values in a second distribution. Table 4 presents p-values and Cliff's deltas.

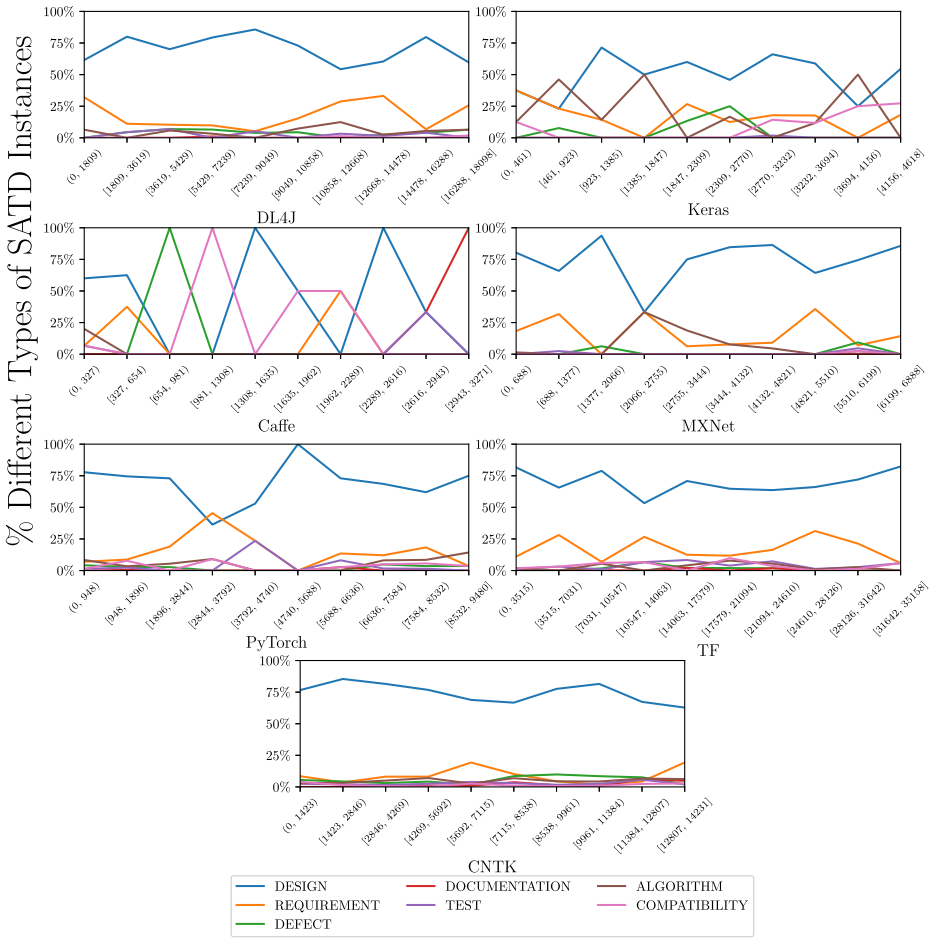

Fig. 2 Distribution of different types of technical debt that are removed in different development phases

Table 4 P-values and Cliff's deltas of the differences between different types of technical debt in terms of their proportions among the removed technical debt instances along the development process

	Algorithm	Compatibility	Defect	Design	Documentation	Requirement
Compatibility						
Defect						
Design	⋆ (+large)	⋆ (+large)	⋆ (+large)			
Documentation	⋆ (-large)	⋆ (-medium)	⋆ (-medium)	⋆ (-large)		
Requirement	⋆ (+large)	⋆ (+large)	⋆ (+large)	⋆ (-large)	⋆ (+large)	
Test	⋆ (-medium)			⋆ (-large)	⋆ (+medium)	⋆ (-large)

We report the pairs of SATD types with p-value < 0.05 (i.e., significant) with ⋆ and present the interpretation of the corresponding Cliff's delta

Then we analyze the removal rate (i.e., the proportion of removed SATD instances among the introduced SATD instances) of different types of technical debt instances at different development phases. For certain development phases, the **removal rate** of different types of technical debt instances is calculated as the proportion of the removed technical debt among the introduced different types of technical debt instances. Figure 3 shows the removal rate of the existing technical debt instances.

To check whether the differences between different types of technical debt in terms of their removal rates among the introduced technical debt instances along the development process are significant, we perform a Kruskal-Wallis H test (1952). Kruskal-Wallis H test is a non-parametric test for comparing whether two or more independent samples originate from the same distribution. As a result, we find that the differences between different types of technical debt in terms of their removal rates in introduced technical debt instances are significant (p-value < 0.05). Then, we perform a Dunn's test with Bonferroni correction to determine which groups differ from each other group (Dunn 1961). Dunn's test can be used for the post-hoc analysis for the specific sample pairs. To calculate the effect size, we calculate the corresponding Cliff's deltas (1993). Cliff's delta is a measure of how often the values in one distribution are larger than the values in a second distribution. Table 5 presents p-values and Cliff's deltas.

Finally, to have an overview of the removal rate of the different types of technical debt in 7 deep learning frameworks before the latest stable release version, we calculate the proportion of removed technical debt instances among all the technical debt instances that are introduced over one year before the latest stable release version. Table 6 shows the removal rate of different types of technical debt before the latest stable release version, as well as the average removal rate for each project. To better view the differences between different types of technical debt, we highlight the removal rates of different types of technical debt which are higher than the corresponding project value in bold.

Results Table 5 shows that the differences between requirement debt and all other types of technical debt except design debt are significant, and the effect sizes range from small (for test debt) to large (for other types of technical debt). Figure 3 shows that the removal rates of requirement debt are one of the highest along the development process, e.g., in Keras, CNTK, Caffe, MXNet, and DL4J. This shows that requirement debt is the most removed technical debt along the development process. Table 4 shows that the differences between requirement debt and other types of technical debt in terms of their proportion among the

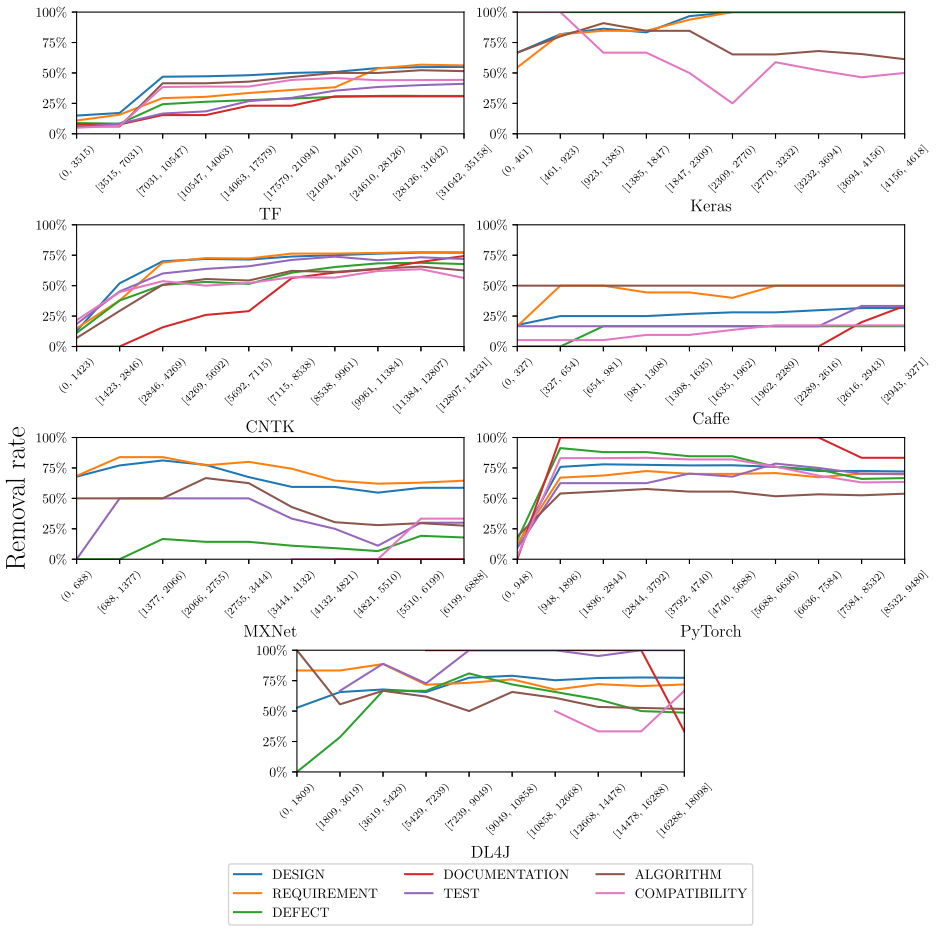

Fig. 3 Removal rate of different types of technical debt along the development process

Table 5 P-values and Cliff's deltas of the differences between different types of technical debt in terms of their removal rates among the introduced technical debt instances along the development process

	Algorithm	Compatibility	Defect	Design	Documentation	Requirement
Compatibility						
Defect						
Design	⋆ (+medium)	⋆ (+large)	⋆ (+medium)			
Documentation				⋆		
Requirement	⋆ (+large)	⋆ (+large)	⋆ (+large)		⋆ (+large)	
Test				⋆ (-small)		⋆ (-small)

We report the pairs of SATD types with p-value < 0.05 (i.e., significant) with ⋆ and present the interpretation of the corresponding Cliff's delta

Table 6 Removal rate of different types of technical debt

Project name	TF	Keras	CNTK	Caffe	MXNet	PyTorch	DL4J	Average
Project level	56.8 %	79.3 %	79.9 %	38.1 %	42.6 %	70.4 %	89.6 %	65.2 %
Design	**68.5 %**	**99.0 %**	**83.3 %**	**38.9 %**	**72.6 %**	**82.9 %**	**93.8 %**	**77.0 %**
Compatibility	**60.1 %**	56.5 %	74.8 %	29.6 %	0.0 %	67.4 %		48.1 %
Defect	51.3 %	50.0 %	**79.6 %**	27.3 %	**61.5 %**	**77.1 %**	74.2 %	60.1 %
Documentation	50.0 %		**86.5 %**	**57.4 %**	0.0 %	**75.0 %**	**100.0 %**	61.5 %
Test	50.0 %	**100.0 %**	78.5 %	28.6 %	**60.0 %**	55.6 %	**93.3 %**	**66.6 %**
Algorithm	**57.6 %**	70.4 %	75.2 %	30.0 %	32.0 %	59.6 %	85.7 %	58.6 %
Requirement	**60.0 %**	**100.0 %**	**81.6 %**	**55.0 %**	**72.1 %**	**75.1 %**	**90.6 %**	**76.3 %**

removed technical debt along the development process are significant, and the effect sizes range from medium to large. Figure 2 shows that requirement debt is resolved in every development phase. More specifically, along the development process, requirement debt has the second largest proportion of removed technical debt along the development process. Developers put the resolution of requirement debt in a high priority. In TensorFlow, Keras, CNTK, and PyTorch, the removal rate of requirement debt increase along with the development. This indicates that developers resolve more requirement debt than the introduction. However, in Caffe, MXNet, and DL4J, the removal rate of requirement debt decrease with fluctuation. This shows that developers introduce more requirement debt than their resolution, and there is an accumulation of requirement debt in Caffe, MXNet, and DL4J. We suggest the project managers should slow down the proposal of new requirements and wait for the resolution of requirement debt. Finally, until the latest stable release version, the removal rates of requirement debt in different frameworks range from 55 % for Caffe to 100 % for Keras.

Table 5 shows that the differences between design debt and all other types of technical debt except requirement debt are significant, and the effect sizes range from negligible (for documentation debt) to large (for compatibility debt). Figure 3 shows that the removal rate of design debt is one of the highest along the development process across the studied deep learning frameworks. This shows that design debt is the second most removed technical debt along the development process. Table 4 shows that the differences between design debt and other types of technical debt in terms of their proportion among the removed technical debt along the development process are significant and large. Figure 2 shows that design debt has the largest proportion among the removed technical debt instances in most of the development phases. This shows that developers paid their effort to the resolution of design debt the most along the development process. For example, in Section 3.1, we observe that the design debt in Caffe is introduced the most at the beginning of the development process and is seldomly introduced after the beginning of the development process. However, in the 5th and 6th development phases, design debt has the largest proportion among the removed technical debt. Though developers seldomly admitted the sub-optimal trade-offs or decisions related to the design of code in the 5th and 6th development phases, they have to pay efforts to the resolution of design debt that is legacy in their past work. In MXNet, though design debt has the largest proportion among the removed technical debt along the development process, the removal rates of design debt decrease with fluctuation after the 3rd development phase. This shows that developers introduce more design debt than removal, and there is an accumulation of design debt. We suggest the developers in MXNet pay more

attention to the design of code. Finally, until the latest stable release version, the removal rates of design debt range from 38.9 % in Caffe to 99.0 % in Keras.

Table 4 shows that the differences between documentation debt and other types of technical debt in terms of their proportion among the removed technical debt along the development process are significant, and the effect sizes range from medium to large; the differences between test debt and algorithm debt, design debt, requirement debt, and documentation debt are significant, and the effect sizes range from medium to large. This shows that **Documentation debt has the smallest proportion among the removed technical debt instances in different development phases, followed by test debt**. Along the development process, we can observe that documentation debt and test debt is removed in certain development phases in CNTK, MXNet, PyTorch, and DL4J. One possible reason is that the number of introduced test debt and documentation debt is small, and limited attention paid by developers can result in a large proportion of test debt and documentation debt get removed. Finally, until the latest stable release version, the removal rates of documentation debt in different frameworks range from 0 % for MXNet to 100 % for DL4J, and the removal rates of documentation debt in different frameworks range from 28.6 % for Caffe to 100 % for Keras.

Requirement debt is removed the most along the development process, followed by design debt. Documentation debt has the smallest proportion among the removed technical debt instances in different development phases, followed by test debt.

3.3 RQ3: Which Types of Technical Debt are Removed the Fastest?

Motivation In Section 3.2, we characterize the removal of different types of technical debt during the development process of different deep learning frameworks. However, it is still unclear about how long does it take to be removed since the introduction of different types of technical debt. In this section, we would like to characterize the removal of different types of technical debt along their lifecycle.

Approach To characterize the removal of different types of technical debt, we perform a series of survival analyses. Survival analysis can statistically analyze the expected duration time of the objects before an event, e.g., death in biological organisms and failure in mechanical systems (Miller 2011). Survival analysis also can handle the case that an object does not have an event during the observation time (i.e., censored). In this paper, survival analysis can characterize the removal of different types of SATD instances. The SATD instances that are not removed before the latest stable release version are right-censored.

To model the time to remove, survival function can give the probability that a SATD instance will survive beyond any specified time (Carpenter 1997). We first estimate the survival function with popular parametric distributions, e.g., Exponential distribution, Weibull distribution, Gamma distribution, Log-Normal distribution, to find their best fit models. Since the underlying data distribution is unknown, we use AIC to compare different models to select the most appropriate model for different types of technical debt in different deep learning frameworks. The Akaike information criterion (AIC) can estimates the quality of each model, relative to each of the other models (McElreath 2020). As a result, we find that the survival time of certain types of technical debt in some deep learning frameworks cannot be significantly fit into any popular parametric distributions (i.e., p-values > 0.05). This motivates us to estimate the survival function with the Kaplan-Meier estimator (Kaplan and

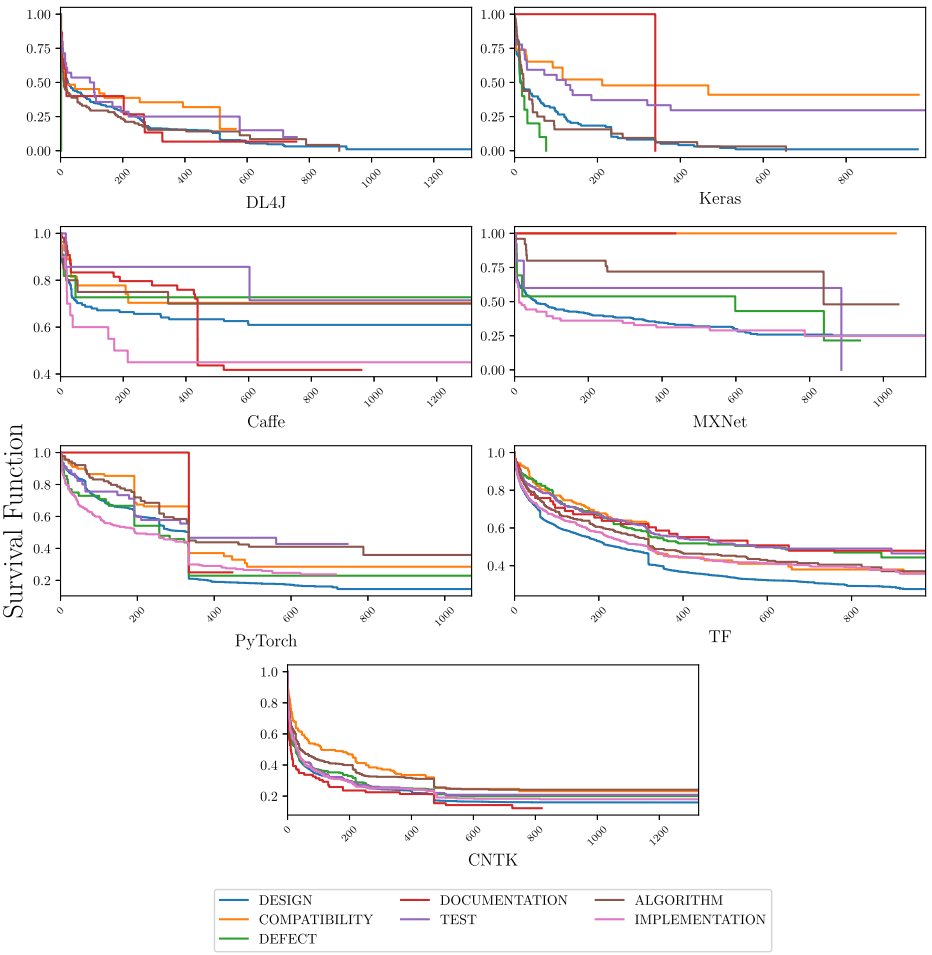

Fig. 4 Survival function of different types of SATD instances in all deep learning frameworks

Meier 1958). The Kaplan-Meier estimator is a non-parametric statistic used to estimate the survival function from lifetime data. Figure 4 plots the survival function of different types of SATD instances in different deep learning frameworks. Table 7 presents the median survival time (i.e., half-life) of different types of technical debt in different frameworks. To better view the differences between different types of technical debt, we highlight the median survival time of different types of technical debt which are shorter than the corresponding project value in bold.

To describe the effect of different types of technical debt on the removal of SATD instances, we regress the types of technical debt against their hazard rate. Hazard rate (i.e., failure rate) is used in survival analysis to describe the number of failures per unit of time. We use Cox's model to estimate the hazard function (a function of time and some covariates that represent the hazard rate). Cox's model is a non-parametric model to estimate the hazard function when the assumption of proportional hazards is true (Cox and Oakes 1984). The proportional hazard assumption is that the shape of the hazard function is the same

Table 7 Half-life of different types of technical debt

Project name	TF	Keras	Caffe	PyTorch	MXNet	CNTK	DL4J	Average
Project	463.0	98.7	202.1	316.3	93.0	40.0	35.9	178.4
Design	**211.3**	**25.7**	**81.3**	**270.0**	**65.2**	**25.6**	31.3	**101.5**
Defect	604.2	**17.1**	**0.2**	**268.2**	INF	**21.8**	INF	182.3
Compatibility	**420.9**	INF		317.6		104.1		280.9
Requirement	**315.8**	**21.3**	477.0	**189.3**	**31.1**	**32.2**	**13.3**	**154.3**
Documentation	660.7		448.8	334.1		**11.3**		363.7
Algorithm	**376.5**	**89.8**	**3.1**	432.3	182.6	50.9	63.2	**171.2**
Test	651.4	339.5	INF	402.5	INF	**33.8**	35.9	292.6

for all individuals and only a scalar multiple changes per individual. We test the proportional hazard assumption using the scaled Schoenfeld residuals implemented in cox.zph() function in R. We find there is no violation of the proportional hazard assumption in the survival time of different types of technical debt instances. Table 8 presents the effects and their p-values of different types of technical debt on the removal of SATD instances. The coefficients measure the impact (i.e., the effect size) of covariates. Hazard ratios (HR) are calculated as the exponential of coefficients. A hazard ratio above 1 indicates a covariate that is positively associated with the event probability, and thus negatively associated with the length of survival.

To check whether the differences in the survival time across the seven technical debt types are statistically significant, we perform a Wei-Lachin Test (1984). Wei-Lachin Test can compare survival distributions of more than two populations. The null hypothesis that different types of technical debt instances have the same removal process. As a result, we find that the difference between different types of technical debt in terms of survival time is significant (p-value < 0.05). Then, we perform a series of Mantel-Cox test as the post-hoc analysis to determine which groups differ from each other group (Mantel 1966). Mantel-Cox Test can compare the survival distributions of two populations. Table 9 presents p-values.

Results Table 9 shows that the differences between design debt and all other types of technical debt are significant. Table 8 shows a strong relationship between the design debt and shorter survival time, i.e., compared with comments that are not identified as SATD instances, design debt is removed 1.56 times faster. This shows that **design debt is removed the fastest along the development process**. Figure 4 shows that along the lifecycle of design debt in different deep learning frameworks, design debt is one of the fastest removed technical debt. In TensorFlow, design debt is removed the fastest along the lifecycle. Table 7 shows that the half-life of design debt range from 25.6 days (for CNTK) to 270.0 days (for

Table 8 Coefficients and hazard ratios of different types of SATD in Cox's model

	design	requirement	defect	documentation	test	algorithm	compatibility
coef	0.44	0.23	0.31	0.12	0.01	0.08	−0.08
HR	1.56	1.26	1.36	1.13	1.01	1.09	0.93
p	⋆	⋆	⋆			⋆	

We report the coefficients with p-value < 0.05 (i.e., significant) with ⋆

Table 9 P-values between different types of SATD in terms of the differences in survival time across the seven technical debt types

	Algorithm	Compatibility	Defect	Design	Documentation	Requirement
Compatibility	⋆					
Defect	⋆	⋆				
Design	⋆	⋆	⋆			
Documentation		⋆		⋆		
Requirement	⋆	⋆		⋆		
Test			⋆	⋆		⋆

We report the pairs of SATD types with p-value < 0.05 (i.e., significant) with ⋆

PyTorch) with an average of 101.5. Developers commonly put the resolution of design debt in the highest priority.

Table 9 shows that the differences between defect debt and algorithm debt, compatibility debt, design debt, and test debt are significant. Table 8 shows a strong relationship between the defect debt and shorter survival time, i.e., compared with comments that are not identified as SATD instances, defect debt is removed 1.36 times faster. This shows that defect debt is the second-fastest removed technical debt along the development process. Figure 4 shows that defect debt is one of the fastest removed technical debt. In Keras and DL4J, defect debt is removed the fastest. Table 7 shows that the half-life of defect debt range from 0.2 days (for CNTK) to 270.0 days (for PyTorch) with an average of 101.5. This shows that developers in Keras and DL4J put the resolution of defect debt in the highest priority.

Table 9 shows that the difference between compatibility debt and all other types of SATD except test debt is significant. Table 8 shows there is no difference between compatibility debt and the comments that are not identified as SATD instances. This shows that **compatibility debt is the slowest removed**. We will discuss the compatibility debt in Section 4.2.

Following that, Table 9 shows that test debt is significantly different from defect debt, design debt, and requirement debt; documentation debt is significantly different from compatibility debt and design debt. Table 8 shows there is no significant difference between documentation debt and the comments that are not identified as SATD instances; there is no difference between test debt and the comments that are not identified as SATD instances. This shows that **documentation debt and test debt is removed slow**. The slow removal pace of documentation indicates that developers put the solving of documentation debt in low priority. The slow removal pace of test debt can be associated with the unaccomplishment of the implementation of related functions to be tested. We will discuss the slow removal of documentation debt and test debt in Section 4.2.

Design debt is removed the fastest, followed by defect detb. Compatibility debt is removed the slowest, followed by test debt and documentation debt.

3.4 RQ4: Who Removes Different Types of Technical Debt?

Motivation Different from the technical debt that is hidden in code, SATD is a kind of technical debt that is acknowledged by the developers who write down the comments. Previous work has illustrated that most of the SATD instances are self-removed, i.e., removed by

the developers who introduce the comments (Maldonado et al. 2017). Moreover, technical debt also can be removed by other developers with more or fewer activities in the project. More specifically, developers with more activities in the project are more likely to accomplish the more difficult programming tasks since they are more familiar with the project, while developers with fewer activities in the project are more likely to accomplish easier programming tasks since they contribute less to the project. However, it is still unclear who removes the different types of technical debt the most during the development of deep learning frameworks.

Approach To do so, we compare the author names and email addresses of the versions that introduce and remove the SATD instances to see if they are the same or not. If the author's names and email addresses are the same in the version that introduces the SATD instance and the version that removes the SATD instance, the SATD instance is self-removed. Otherwise, the SATD instance is removed by other developers. Since there is the risk of misclassifying the authors that change their names in the source code repository during the evolution of the project, we rely on Open-hub's data to merge developer identities. Hence, our study is only as accurate as Open-hub's classification.

If a SATD instance is removed by other developers, we compare the remover's activities in the project with the introducer's activities in the project at the removal time point. We measure a developer's activities in the project by the number of commits performed by that developer in the given project. More specifically, if the number of commits done by remover is more than the number of commits done by introducer at the removal of the SATD, then we consider the SATD to be removed by the developers with more activities in the project; otherwise, the SATD is removed by developers with fewer activities in the project.

Table 10 presents the proportion of technical debt that is removed by developers with different activities in the project for different types of technical debt in different frameworks. Concretely, we present the proportion of technical debt that is removed by other developers with more activities in the project (shortened as MORE), the proportion of technical debt that is removed by other developers with fewer activities in the project (shortened as FEWER), and the proportion of self-removed technical debt.

To check the differences between different developers (i.e., developers with more activities in the project, developers with fewer activities in the project, and developers who introduce the SATD instances) in terms of their proportion among the removed SATD instances are significant for each type of technical debt, we perform seven Kruskal-Wallis H tests (1952). Kruskal-Wallis H test is a non-parametric test for comparing whether two or more independent samples originate from the same distribution. As a result, we find that the difference between different developers in terms of their proportion among the removed SATD instances are significant (p-value < 0.05). Then, we perform a Dunn's test with Bonferroni correction to determine which groups differ from each other group (Dunn 1961). Dunn's test can be used for the post-hoc analysis for the specific sample pairs. To calculate the effect size, we calculate the corresponding Cliff's deltas (1993). Cliff's delta is a measure of how often the values in one distribution are larger than the values in a second distribution. Table 11 presents p-values and Cliff's deltas.

To check the differences between different types of technical debt instances in terms their proportion that is removed by different developers (i.e., developers with more activities in the project, developers with fewer activities in the project, and developers who introduce the SATD instances) among the removed SATD instances are significant, we perform three Kruskal-Wallis H tests (1952). Kruskal-Wallis H test is a non-parametric test for comparing whether two or more independent samples originate from the same distribution. As a

Table 10 Proportion of technical debt removed by developers with different activities in the project for each type

Type	Removal type	TF	Keras	CNTK	Caffe	MXNet	PyTorch	DL4J	Average
DESIGN	Self	32.9 %	50.7 %	36.9 %	59.2 %	39.0 %	80.9 %	54.9 %	50.6 %
	More	45.3 %	31.0 %	39.0 %	30.6 %	44.2 %	5.9 %	33.1 %	32.7 %
	Fewer	21.8 %	18.3 %	24.1 %	10.2 %	16.9 %	13.2 %	12.0 %	16.6 %
COMPATIBILITY	Self	17.6 %	0.0 %	21.0 %	50.0 %		87.7 %		35.3 %
	More	69.2 %	100.0 %	58.0 %	37.5 %		8.8 %		54.7 %
	Fewer	13.2 %	0.0 %	21.0 %	12.5 %		3.5 %		10.0 %
DEFECT	Self	32.1 %	100.0 %	20.5 %	66.7 %	14.3 %	78.1 %	41.2 %	50.4 %
	More	56.8 %	0.0 %	42.6 %	33.3 %	85.7 %	15.6 %	41.2 %	39.3 %
	Fewer	11.1 %	0.0 %	36.9 %	0.0 %	0.0 %	6.3 %	17.6 %	10.3 %
DOCUMENTATION	Self	38.9 %		22.6 %	19.4 %		100.0 %		45.2 %
	More	44.4 %		38.7 %	67.7 %		0.0 %		37.7 %
	Fewer	16.7 %		38.7 %	12.9 %		0.0 %		17.1 %
TEST	Self	41.8 %	100.0 %	42.0 %	50.0 %	33.3 %	77.3 %	46.2 %	55.8 %
	More	38.8 %	0.0 %	30.6 %	0.0 %	33.3 %	9.1 %	46.2 %	22.6 %
	Fewer	19.4 %	0.0 %	27.4 %	50.0 %	33.3 %	13.6 %	7.7 %	21.6 %
ALGORITHM	Self	33.3 %	42.1 %	29.8 %	66.7 %	28.6 %	70.0 %	52.4 %	46.1 %
	More	49.1 %	47.4 %	43.0 %	33.3 %	71.4 %	10.0 %	33.3 %	41.1 %
	Fewer	17.6 %	10.5 %	27.2 %	0.0 %	0.0 %	20.0 %	14.3 %	12.8 %
IMPLEMENTATION	Self	41.5 %	54.5 %	37.3 %	30.0 %	51.4 %	71.3 %	61.2 %	49.6 %
	More	43.9 %	9.1 %	40.3 %	30.0 %	32.4 %	2.3 %	28.6 %	26.7 %
	Fewer	14.6 %	36.4 %	22.4 %	40.0 %	16.2 %	26.4 %	10.2 %	23.7 %

result, we find that the difference between different types of technical debt in terms of their proportion that is removed by different developers is significant (p-value < 0.05). Then, we perform a Dunn's test with Bonferroni correction to determine which groups differ from each other group (Dunn 1961). Dunn's test can be used for the post-hoc analysis for the specific sample pairs. To calculate the effect size, we calculate the corresponding Cliff's deltas (1993). Cliff's delta is a measure of how often the values in one distribution are larger than the values in a second distribution. Tables 12, 13, and 14 present p-values and Cliff's deltas.

Results Table 11 shows that the differences between the developers who introduce the SATD instances and other developers are significant and small in test debt. Table 14 shows

Table 11 P-values and Cliff's deltas of the differences between different developers who removed the SATD instances for each type of technical debt

	Algorithm	Compatibility	Defect	Design	Documentation	Requirement	Test
Fewer - More	⋆-small	⋆-medium	⋆-small	⋆-small	⋆-small	⋆-small	⋆
Fewer - Self	⋆-small	⋆-small		⋆-small		⋆-small	⋆-small
More - Self	⋆	⋆+small	⋆+small	⋆	⋆+small	⋆	⋆-small

We report the pairs of SATD types with p-value < 0.05 (i.e., significant) with ⋆ and present the interpretation of the corresponding Cliff's delta

Table 12 P-values and Cliff's deltas of the differences between different types of technical debt in terms of whether the SATD instances are removed by developers with fewer activities in the project

	Algorithm	Compatibility	Defect	Design	Documentation	Requirement
Compatibility	⋆					
Defect	⋆	⋆ (+small)				
Design		⋆	⋆			
Documentation						
Requirement			⋆			
Test		⋆				

We report the pairs of SATD types with p-value < 0.05 (i.e., significant) with ⋆ and present the interpretation of the corresponding Cliff's delta

that in terms of whether SATD instances are removed by developers who introduce them, documentation debt is significantly different from design debt, requirement debt, and test debt, and the effect sizes are small; defect debt is significantly different from design debt, requirement debt, and test debt, and the effect sizes are small. This shows that documentation debt and defect debt is the least self-removed. Test debt, design debt, and requirement debt are the most self-removed. This shows that the developers who introduce the test debt, design debt, and requirement debt acknowledge the existence of the introduced technical debt. They paid off these technical debt instances in their future work. In contrast, the documentation debt and defect debt are removed the least by the developers who introduced them. Table 11 shows that the differences between developers with more activities in the project and other developers are significant and small in defect debt and documentation debt, indicating that documentation debt and defect debt are removed more developers with more activities in the project. One possible reason is that developers who introduce the defect debt and documentation debt may not know how to resolve these technical debt instances.

Table 11 shows that the differences between the developers with fewer activities in the project and others who removed the SATD instances are significant and small in algorithm debt, compatibility debt, design debt, and requirement debt. Table 12 shows that in terms of whether SATD instances are removed by developers with fewer activities in the project, the

Table 13 P-values and Cliff's deltas of the differences between different types of technical debt in terms of whether the SATD instances are removed by developers with more activities in the project

	Algorithm	Compatibility	Defect	Design	Documentation	Requirement
Compatibility	⋆					
Defect						
Design	⋆	⋆ (-small)	⋆			
Documentation				⋆ (+small)		
Requirement		⋆ (-small)	⋆		⋆ (-small)	
Test	⋆	⋆	⋆		⋆ (-small)	

We report the pairs of SATD types with p-value < 0.05 (i.e., significant) with ⋆ and present the interpretation of the corresponding Cliff's delta

Table 14 P-values and Cliff's deltas of the differences between different types of technical debt in terms of whether the SATD instances are removed by developers who introduce the SATD instances

	Algorithm	Compatibility	Defect	Design	Documentation	Requirement
Compatibility						
Defect	⋆					
Design	⋆		⋆ (+small)			
Documentation				⋆ (-small)		
Requirement	⋆	⋆	⋆ (+small)		⋆ (+small)	
Test	⋆	⋆	⋆ (+small)		⋆ (+small)	

We report the pairs of SATD types with p-value < 0.05 (i.e., significant) with ⋆ and present the interpretation of the corresponding Cliff's delta

difference between compatibility debt and defect debt is significant and small. This shows that compatibility debt is removed the least by the developers with fewer activities in the project.

Table 11 shows that the differences between the developers who introduce the SATD instances and the developers with more activities in the project are significant and small in compatibility debt, defect debt, documentation debt, and test debt. Table 13 shows that compatibility debt is significantly different from design debt, and requirement debt, and the effect sizes are small; documentation debt is significantly different from design debt, test debt, and requirement debt, and the effect sizes are small. This shows the compatibility debt and documentation debt are the most removed by the developers with more activities in the project. One possible reason for compatibility debt is the resolution of compatibility debt may need the replacement of external dependencies. However, the management of the update of external dependencies may require the privilege of administration in modern software management. Besides, the implementation of the functions provided by external dependencies can require for the developers with more activities in the project. Therefore, compatibility debt is removed the least by the developers with fewer activities in the project but is the most removed by the developers with more activities in the project. One possible reason for documentation debt is that the documentation debt can be difficult for developers to resolve. We will discuss the removal of documentation debt in Section 4.2.

Documentation debt and defect debt is the least self-removed. Test debt, design debt, and requirement debt are the most self-removed. Compatibility debt and documentation debt are the most removed by the developers with more activities in the project.

4 Discussion

In this section, we depict the evolution of the frequencies of different types of technical debt along the development process, present our discussion on the removal patterns of different types of technical debt based on the findings we mentioned above, and provide actionable suggestions for practitioners, project managers, and researchers. Finally, we present threats to validity.

4.1 The Evolution of Frequencies of Different Types of Technical Debt Along the Development Process

In Section 3.1, we investigate the introduction of different types of technical debt along the development process. In Section 3.2, we investigate the removal of different types of technical debt along the development process. However, it is still unclear the evolution of the frequencies of different types of technical debt along the developing process.

We plot the evolution of the frequencies of different types of technical debt along the development process. To characterize the frequencies of different types of technical debt along the development process, we first divide the whole development process into ten **development phases** based on the chronological order of the commits. Then we count the number of different types of technical debt instances in each development phase. Figure 5 shows the frequencies of different types of SATD instances along the developing process in 7 deep learning frameworks. The frequencies of different types of technical debt at different development stages illustrate the challenges which are mainly confronted with by the developers at that time. Ups and downs in the plots along the development process depict the changes in developers' challenges over time.

Keras is an outlier here. At the beginning of the development of Keras, requirement debt is the most common debt. Developers are confronted with the fast iterate of the project and they record the unimplemented tasks. Then the number of design debt instances increases along the development process, and design debt is the most common technical debt since the fourth development phase. Many SATD instances are written down to express their dissatisfaction with the design of the implementation of tasks. Meanwhile, the number of algorithm debt instances increases, and become the second most common technical debt. It shows developers concentrate on the algorithms, i.e., the cutting edge deep learning module and the efficient computation method. And currently, compatibility debt increases step by step and becomes one of the most common debt now. This is because that Keras does not handle low-level operations such as tensor products, convolutions, and so on. Instead, it relies on a specialized, well-optimized tensor manipulation library, e.g., Theano, to serve as the *backend engine* of Keras. This leads to that Keras enjoys the convenience provided by backend projects at the cost of the maintenance of dependencies.

4.2 Implications and Removal Patterns of Different Types of Technical Debt

In this subsection, we discuss how different types of technical debt are removed based on the aforementioned findings. Based on the discussion, we provide suggestions for developers, researchers, and project managers in the resolution of technical debt.

(1) **Design debt:** In Section 3.1, we observe that design debt is the most introduced technical debt along the development process. In Section 3.2, we observe that design debt has the largest proportion among the removed technical debt instances in most of the development phases. In Section 3.3, we observe that design debt is removed the fastest.

(2) **Defect debt:** In Section 3.1, Table 2 shows that the proportion of defect debt is significantly lower than design debt and requirement debt. One possible reason is the wide use of the industrial issue tracking system. In Section 3.4, we observe that defect debt is one of the least self-removed. In Section 3.3, we observe that the removal pace of defect debt is the second-fastest. **We suggest that all developers participate in the resolution of defect debt.**

(3) **Requirement debt:** In Section 3.1, we observe that requirement debt is the second most common technical debt during the development process. This shows that developers

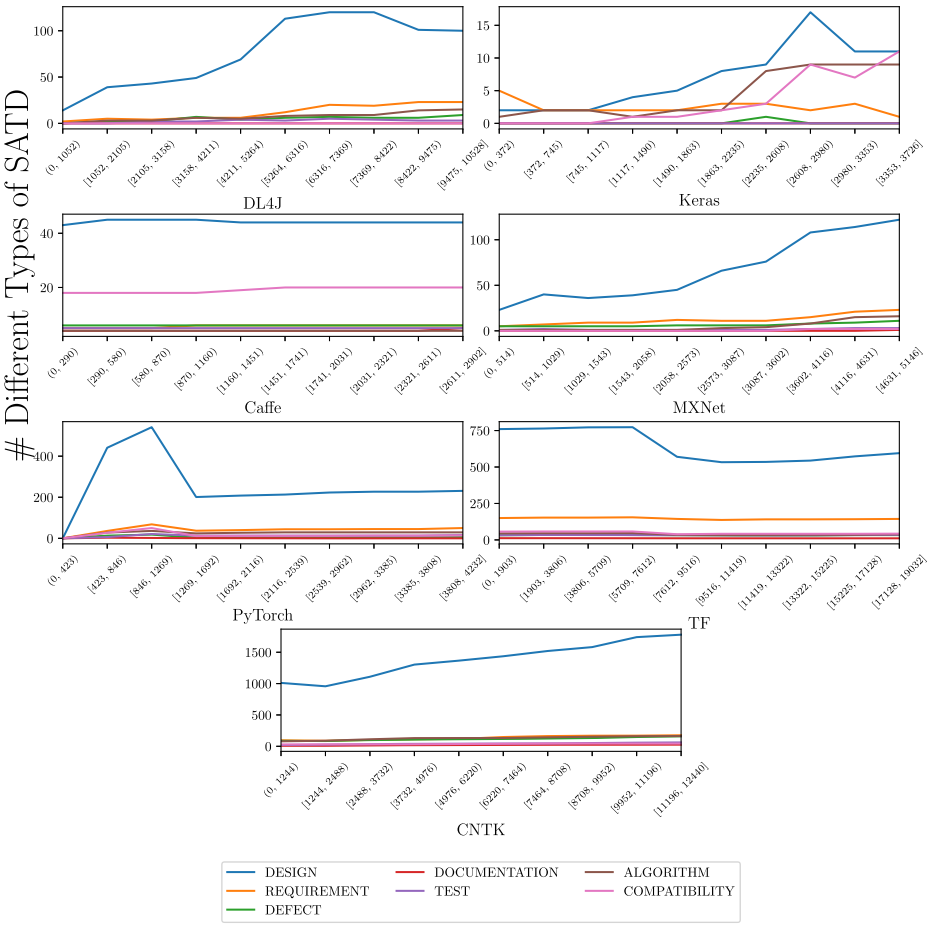

Fig. 5 Evolution of frequencies of different types of technical debt along the development process in different deep learning frameworks

commonly cannot finish the implementation of certain tasks in time. In Section 3.4, we observe that requirement debt is one of the most self-removed technical debt. In Section 3.3, we observe that requirement debt is removed relatively fast.

(4) **Documentation debt:** In Section 3.1, we observe that the documentation debt is the least common technical debt. This shows that developers seldomly admitted the sub-optimal trade-offs or decisions related to the documentation during the development process. In Section 3.4, we observe that documentation debt is the least self-removed technical debt, and is the most removed by the developers with more activities in the project. In Section 3.3, we observe that documentation debt is the second slowest removed technical debt. This shows that compared with other developers, developers with more activities in the project can resolve documentation debt but not timely. For example, the comment in TensorFlow:

> Given a numerical function "f", returns another numerical function "g", such that if "f" takes N inputs and produces M outputs, "g" takes N + M inputs and produces N outputs. I.e., if $(y_1, y_2, ..., y_M) = f(x_1, x_2, ..., x_N)$, g is a function

which is $(dL/dx_1, dL/dx_2, ..., dL - /dx_N) = g(x_1, x_2, ..., x_N, dL/dy_1, dL/-dy_2, ..., dL/dy_M)$, where L is a scalar-value function of $(...x_i...)$. TODO(zhifengc): Asks math expert to say the comment again.

This comment shows that the documentation debt waits for the experts' resolution. We suggest that project managers allocate the documentation debt timely to the developers with more activities in the project.

(5) **Test debt:** In Section 3.1, we observe that the test debt is one of the least common technical debt. One possible reason is the widespread use of industrial test systems. In Section 3.4, we observe that test debt is the most self-removed technical debt. In Section 3.3, we observe that test debt is removed slower than other types of technical debt. One reason is that the removal of test debts is associated with the accomplishment of the requirement to be tested. For example, in a comment in PyTorch:

TODO(jiayq): when there are backward and GPU implementations, enablethese two.self.assertDeviceChecks(dc, op, [X, scale, bias], [0])self.assertGradientChecks(gc, op, [X, scale, bias], 0, [0])

This shows that some requirement debt instances are not presented in a form of requirement debt but in the form of test debt. We suggest that project managers can check the unaccomplished requirement from the test debt.

(6) **Compatibility debt:** In Section 3.4, we observe that compatibility debt is removed the least by developers with fewer activities in the project, but is removed the most by developers with more activities in the project. However, in Section 3.3, we observe that the removal of compatibility debt is the slowest. One possible reason is that the removal of compatibility debts is associated with the release of qualified dependencies or the implementation of related functions. For example, there is a comment in PyTorch:

XXX: Gloo does not support scatter/gather/reduce

This comment indicates that the current code using Gloo[23] to implement related tasks is sub-optimal. However, this comment is not removed before the latest stable release version (i.e., May 31, 2018). This is because that Gloo began to implement "scatter/gather/reduce" related code since Oct 2018,[24] which is lagged behind the latest stable release version of PyTorch. Dependencies that different projects rely on evolve at different paces, some of them remain unqualified until now. We suggest developers re-implement specific functions within the broader system architecture (Sculley et al. 2015). The re-implementation of the dependencies can make the frameworks not bind themselves tightly with its dependencies.

(7) **Algorithm debt:** In Section 3.1, we observe that the algorithm debt is the third most common technical debt. This shows that for the development of deep learning frameworks, the sub-optimal trade-offs or decisions related to the algorithms are common. In Section 3.4, Table 11 shows that algorithm debt is significantly less removed by developers with fewer activities in the project. In Section 3.3, Tables 8 and 9 shows that there is no significant difference between documentation debt and test debt, which is one of the slowest removed technical debt. One possible reason is that the algorithms in deep learning frameworks are of a wide variety and still advancing, and newly proposed algorithms may be out of the range of developers' skill and difficult to implement. For example, in a comment in TensorFlow:

[23] https://github.com/facebookincubator/gloo

[24] https://github.com/facebookincubator/gloo/commits/1d9e62aff9d7143129a69c8eb23e8351-e686ff3a/gloo/scatter.cc

> Returns Poisson-distributed random number. Uses Knuth's algorithm. Take care: this takes time proportional to lambda. Faster algorithms exist but are more complex.

This shows that developers are aware of a faster algorithm to finish the development tasks, but they refuse to implement the algorithm due to complexity concerns. We suggest that all developers are involved in the fixing of algorithm debt.

For **project managers**, Section 4.1 depicts the evolution of the frequencies of different types of technical debt in different development phases along the development process. These evolutions can reflect the changes of developers' concern at different stages along the development process. For example, Section 3.2 shows that though design debt has the largest proportion among the removed technical debt along the development process in MXNet, the removal rates of design debt decrease with fluctuation after the 3rd development phase. This shows that developers introduce more design debt than removal, and there is an accumulation of design debt. Project managers should find a balance between the quality of the project and the proposal of new requirements. We suggest project managers take the evolution of the frequencies of SATD into consideration when managing their projects.

For **researchers**, we encourage future researchers to investigate the differences in the removal of the unresolved defects between the ones in the issue tracking system with the ones that are admitted as technical debt. Our research finds that though issue tracking systems are used in the development process, there still is test debt and defect debt in source code. However, it is unclear whether the unresolved defects attract less developer attention as compared to the ones in those issue tracking system systems.

We also encourage researchers to survey and categorize developers' intentions in resolving different types of technical debt. In Section 3.3, our findings suggest that design debt is removed the fastest compared to all other types of technical debt, compatibility debt is removed the slowest. However, the underlying reasons for developers to remove different types of technical debt in different paces are still uncleared. We suggest further studies could survey and categorize developers' intentions when they resolving different types of technical debt.

4.3 Threats to Validity

Threats to internal validity concern factors that could have influenced our results. To identify SATD in a project, we use source code comments that describe part of the source code containing technical debt. One threat of using source code comments is the consistency of changes between the comments and the code, i.e., in some cases the comment may change but not the code and vice versa. However, previous work showed that between 72–91 % of the code and comment changes are consistent, i.e., code and comments co-change together (Potdar and Shihab 2014).

To avoid bias caused by the SATD instances that are introduced recently before the latest stable version (i.e., right censoring) (Quesenberry et al. 1989), we exclude the SATD instances that are introduced in one year before the latest stable release version. One threat is that the SATD instances that are introduced most recently before the pruning-out window could still be subject to the right censoring problem and are not removed yet. However, our findings show that 75 % of the SATD instances that are introduced before the latest stable version are removed in 299 days at the most (for PyTorch). We believe only a small proportion of SATD instances that are introduced most recently before the pruning-out window are not removed.

To classify the detected source code comments into different types, we heavily depended on a manual process. Like any human activity, our manual classification is subject to personal bias and subjectivity. To reduce personal bias in manual classification of code comments, as we indicate in Section 2.4, the first author randomly sampled a statistically representative sample of 1,000 SATD instances from the 29,778 detected SATD instances using a 95 % confidence level with a 10 % confidence interval. We invite an independent Ph.D. student, who is not an author of this paper, to manually classify the randomly sampled 1,000 SATD instances. The most common disagreement is that one technical debt instance can be associated with more than one category. For example, in a comment in TensorFlow:

> This isn't strictly correct since in ghost batch norm, you are supposed to sequentially update the moving_mean and moving_variancewith each sub-batch. However, since the moving statistics are only used during evaluation, it is more efficient to just update in one step and should not make a significant difference in the result.

The first author labels this comment as an algorithm debt instance as this comment shows that the current implementation is a "more efficient" workaround. In contrast, the independent Ph.D. student supposes this comment as a defect debt instance since this comment shows that the current implementation "is not strictly correct" in certain cases. In fact, this SATD instance can be associated with two categories based on the interpretation of different people and can be labeled as defect debt and algorithm debt. This shows that the labeling process is subject to personal bias and subjectivity. For a certain technical debt instance, if we only focus on its introduction and removal as a certain category, there would be fewer SATD instances for other categories. However, a high level of agreement between the classification given by the Ph.D. student and the first author is reported with Cohen's kappa coefficient of $+0.79$. This gives us high confidence in the dataset used in our paper.

To identify the introduction and the removal of SATD instances, we consider the source code comments that do not exist anymore in a source code file as the removal of SATD. The file where the comment exists being deleted also indicates that the comment does not exist. However, in some cases, source code is partly moved from one file to another. We treat this case as the removal of SATD in the original file and the introduction of SATD in the target file.

Moreover, we compare the author's names and email addresses to see if the SATD instances are self removed. However, the authors can change their names in the source code repository during the evolution of the project. To mitigate this threat, we rely on Open-hub's data to merge developer identities. Hence, our study is only as accurate as Open-hub's classification.

Threats to external validity concern the generalization of our findings. Our study is conducted on seven large open source deep learning frameworks. Though we have discussed the similarities and differences between the deep learning frameworks and prior studies, our findings may not be generalized to other open source or commercial projects. In the future, we will analysis SATD in other systems.

We only discuss the test debt and the defect debt that are recorded in the source code. However, issue tracking systems are used in the development process: our research does not consider the test debt and defect debt in issue tracking systems. In the future, we plan to compare the test debt and defect debt reported in source code with that reported by means of issue tracking systems.

5 Related Work

We divide our related work into two parts: the works on software engineering for deep learning and the research works on technical debt We also compare the removal of technical debt in deep learning frameworks at the project-level with that in prior studies.

5.1 Software Engineering for Deep Learning

Considering the popularity and the importance of the deep learning projects, many researchers focus on developing solutions to help better engineer deep learning systems and libraries.

Many previous work focuses on the test of deep learning projects (Sun et al. 2018a, b; Ma et al. 2018; Zhang et al. 2018a). For example, Pei et al. (2017) propose DeepXplore to systematically test DL systems and automatically identify erroneous behaviors without manual labels. Tian et al. (2018) propose DeepTest to automatically test DNN-driven autonomous cars, which can use test images that generated by different realistic transformations like rain, fog and lighting conditions.

Besides works focusing on testing of deep learning projects, Zhang et al.'s work (2018b) studies the characteristics of deep learning defects. They study the TensorFlow application bugs from Stack Overflow and Github, and find the root causes of the defects, e.g., incorrect model parameter or structure. Islam et al. (2019) investigate the bugs related to the five popular deep learning libraries, i.e., Caffe, Keras, TensorFlow, Theano, and Torch and find that data bug and logic bug are the most severe bug types. Moreover, Sculley et al. (2015) empirically summarize the technical debt in machine learning systems during their development. They explore several ML-specific risk factors in deep learning project design, including boundary erosion, entanglement, hidden feedback loops, undeclared consumers, data dependencies, and so on.

Different from those research works, our work inspects deep learning frameworks from another perspective: the removal of different types of technical debt.

5.2 Technical Debt

Cunningham (1993) introduce the metaphor, technical debt, to describe the consequences of poor software development. After the proposal of the metaphor, many researchers focus on how technical debt has been used to communicate the issues that developers find in the code in a way that managers can understand (Seaman and Guo 2011; Kruchten et al. 2013; Brown et al. 2010; Lim et al. 2012).

Potdar et al.'s work (2014) uses comments to identify technical debt and nominates such technical debt as **self-admitted technical debt**. They find that self-admitted technical debt is common cross projects and 2.4–31 % of the files in four traditional application projects contain such debt. Bavota and Russo (2016) replicate the study of Potdar et al.'s work (2014) on a large set of traditional application projects, i.e., Apache and Eclipse projects and find that approximately 57 % of self-admitted technical debts get removed and around 63 % of SATD are self-removed. Maldonado et al.'s work (2017) inspects the introduction and removal of self-admitted technical debt in five open source traditional application projects and find that the majority of self-admitted technical debt is removed (74.4 % on average) in 82–613.2 days on average. Zampetti et al. investigated whether SATD is "accidentally"

removed, and the extent to which the SATD removal is being documented (Zampetti et al. 2018). They observed that 8 % of the SATD removal is acknowledged in commit messages, and most of the changes addressing SATD require complex source code changes.

Many previous works also study the negative impact of technical debt during the development of projects. Zazworka et al.'s work (2011) conducts a study to measure the impact of technical debt on software quality. They find that god classes are more likely to change, and therefore, have a higher impact in software quality. Fontana et al.'s work (2012) investigates design technical debt and propose an approach to classify which code smell should be addressed first. Wehaibi et al.'s work (2016) finds that self-admitted technical debt leads to more complex changes in the future development process.

Many previous works also investigate the management of technical debt during the development of projects. For example, de Almeida et al. proposed a framework for the prioritization of technical debt using a business-driven approach built on top of business processes (de Almeida et al. 2018). They also interviewed a set of IT business stakeholders, and collected and analyzed different sets of technical debt items, comparing how these items would be prioritized using a purely technical versus a business-oriented approach (de Almeida et al. 2018). Zampetti et al. built a multi-level classifier capable of recommending six SATD removal strategies, e.g., changing API calls, conditionals, method signatures, exception handling, return statements, or telling that a more complex change is needed (Zampetti et al. 2020).

Moreover, the metaphor, technical debt, has been gradually extended to different types, e.g., design (Lim et al. 2012), and even documentation (Seaman and Guo 2011), requirements (Ernst 2012), and testing (Shull 2011). The work most relevant to us is Maldonado and Shihab's work (2015), where they manually analyze the comments of 5 open source traditional application projects. They find that there are five types of technical debt that are admitted in comments: design debt, defect debt, documentation debt, requirement debt and test debt.

Compared to these research works, our research focus on the removal of different types of technical debt in a family of software systems, i.e., the development of deep learning frameworks, where different frameworks are expected to achieve a same goal (i.e., offering high-level programming interfaces to deep learning applications) with the implementation of concrete tasks (e.g., implement core building blocks for designing, training and validating deep neural networks). This enable us to find common patterns on the removal of technical debt.

5.3 Comparison with Prior Studies

Our research investigates technical debt in deep learning frameworks by analyzing the SATD in 7 open-source deep learning frameworks. However, previous research (Maldonado et al. 2017) also investigates the removal of SATD on 5 open source traditional applications projects, i.e., Camel, Gerrit, Hadoop, Log4j, and Tomcat. However, compared with our work, they perform an empirical study on the removal rate, survival time, and self-removal rate of the studies project at the project level rather than the type level. Therefore, our study is not a replication study on another set of projects. To compare the removal of SATD in deep learning frameworks with that in prior studies, we present their findings as well as the findings in our research in Table 15.

To check whether the differences between the removal of technical debt in deep learning frameworks and that in prior studies are statistically significant, we perform a Mann-

Table 15 Comparison between our findings and prior studies

Topic	Prior study	Our study
Proportion of removed SATD	74.4 % of SATD comments are removed on average. Maldonado et al. (2017)	67 % of the identified SATD comments is removed
Survival time of removed SATD	from 18.2 to 172.8 days on median. Maldonado et al. (2017)	from 9 to 95 days on median
Proportion of self-removed SATD	54.4 % of SATD comments are self-removed (Maldonado et al. 2017)	the average self-remove rate is 42.19 %

Whitney U test (1947). As a result, the mean survival time of the technical debt in the deep learning frameworks is significantly shorter than that in prior studies (p-value < 0.05). This shows that technical debt in deep learning frameworks is removed faster than the technical debt in traditional applications that is studied in prior work. Developers of deep learning frameworks are more active and put the solution of technical debt in higher priority compared with developers in traditional applications. One possible reason is that a framework has a large user base, which puts greater pressure on the removal of technical debt.

Moreover, as indicated in Table 15, compared with prior studies, SATD instances are removed relatively less in deep learning frameworks. One possible interpretation is that developers of traditional applications put the resolution of technical debt in a relatively higher priority than the developers of deep learning frameworks. Another possible interpretation is that technical debt in traditional applications are more easily resolved than deep learning frameworks. In the future, we plan to compare the effort cost of the resolution technical debt in different types of projects (e.g., deep learning, traditional, IoT). Furthermore, over half of the SATD is self-removed in traditional applications while less than half of the SATD is self-removed in deep learning frameworks. This shows that the resolution of the technical debt in deep learning frameworks involve more developers

6 Conclusion

In this paper, we inspect the removal of different types of technical debt by mining SATD in the history version of 7 open source deep learning framework projects. As a result, we find that developers admit design debt the most, and the removal rate of requirement debt is significantly higher than other types of technical debt. Design debt is removed the fastest among all the types of technical debt, while compatibility debt is removed the slowest. Documentation debt and defect debt is the least self-removed. Test debt, design debt, and requirement debt are the the most self-removed. Compatibility debt and documentation debt are the most removed by the developers with more activities in the project. Based on these findings, we depict the evolution of the frequencies of different types of technical debt along the development process. In the future, we will examine the introduction and removal of technical debt with other evidence, such as by an interview.

Acknowledgements This research was partially supported by the National Key R&D Program of China (No. 2018YFB1003904) and the Australian Research Council's Discovery Early Career Researcher Award (DECRA) (DE200100021).

References

Al-Qizwini M, Barjasteh I, Al-Qassab H, Radha H (2017) Deep learning algorithm for autonomous driving using googlenet. In: 2017 IEEE intelligent vehicles symposium (IV). IEEE, pp 89–96

Alves NS, Ribeiro LF, Caires V, Mendes TS, Spínola RO (2014) Towards an ontology of terms on technical debt. In: Sixth international workshop on managing technical debt (MTD), 2014. IEEE, pp 1–7

Bavota G, Russo B (2016) A large-scale empirical study on self-admitted technical debt. In: IEEE/ACM 13th working conference on mining software repositories (MSR), 2016. IEEE, pp 315–326

Brown N, Cai Y, Guo Y, Kazman R, Kim M, Kruchten P, Lim E, MacCormack A, Nord R, Ozkaya I, et al. (2010) Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP workshop on future of software engineering research. ACM, pp 47–52

Carpenter M (1997) Survival analysis: a self-learning text

Cliff N (1993) Dominance statistics: ordinal analyses to answer ordinal questions. Psychol Bull 114(3):494

Cox DR, Oakes D (1984) Analysis of survival data, vol 21. CRC Press

Cunningham W (1993) The wycash portfolio management system. ACM SIGPLAN OOPS Messenger 4(2):29–30

de Almeida RR, Kulesza U, Treude C, Lima AHG et al (2018) Aligning technical debt prioritization with business objectives: a multiple-case study. In: 2018 IEEE international conference on software maintenance and evolution (ICSME), IEEE, pp 655–664

Dunn OJ (1961) Multiple comparisons among means. J Am Stat Assoc 56(293):52–64

Ernst NA (2012) On the role of requirements in understanding and managing technical debt. In: Proceedings of the third international workshop on managing technical debt, IEEE Press, pp 61–64

Ernst NA, Bellomo S, Ozkaya I, Nord RL, Gorton I (2015) Measure it? Manage it? Ignore it? Software practitioners and technical debt. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering, ACM, pp 50–60

Fontana FA, Ferme V, Spinelli S (2012) Investigating the impact of code smells debt on quality code evaluation. In: Proceedings of the third international workshop on managing technical debt, IEEE Press, pp 15–22

Goodfellow IJ, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, Courville A, Bengio Y (2014) Generative adversarial networks. arXiv:1406.2661 [cs, stat]

Huang Q, Shihab E, Xia X, Lo D, Li S (2018) Identifying self-admitted technical debt in open source projects using text mining. Empir Softw Eng 23(1):418–451

Huval B, Wang T, Tandon S, Kiske J, Song W, Pazhayampallil J, Andriluka M, Rajpurkar P, Migimatsu T, Cheng-Yue R, et al. (2015) An empirical evaluation of deep learning on highway driving. arXiv:150401716

Ioffe S, Szegedy C (2015) Batch normalization: accelerating deep network training by reducing internal covariate shift. arXiv:1502.03167 [cs]

Islam MJ, Nguyen G, Pan R, Rajan H (2019) A comprehensive study on deep learning bug characteristics. In: ESEC/FSE'19: the ACM joint european software engineering conference and symposium on the foundations of software engineering (ESEC/FSE), ESEC/FSE 2019

Jouppi NP, Young C, Patil N, Patterson D, Agrawal G, Bajwa R, Bates S, Bhatia S, Boden N, Borchers A, et al. (2017) In-datacenter performance analysis of a tensor processing unit. In: 2017 ACM/IEEE 44th annual international symposium on computer architecture (ISCA), IEEE, pp 1–12

Kaplan EL, Meier P (1958) Nonparametric estimation from incomplete observations. J Am Stat Assoc 53(282):457–481

Klinger T, Tarr P, Wagstrom P, Williams C (2011) An enterprise perspective on technical debt. In: Proceedings of the 2nd Workshop on managing technical debt, ACM, pp 35–38

Kruchten P, Nord RL, Ozkaya I, Falessi D (2013) Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. ACM SIGSOFT Software Engineering Notes 38(5):51–54

Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. J Am Stat Assoc 47(260):583–621

Li Z, Avgeriou P, Liang P (2015) A systematic mapping study on technical debt and its management. J Syst Softw 101:193–220

Lim E, Taksande N, Seaman C (2012) A balancing act: what software practitioners have to say about technical debt. IEEE Softw 29(6):22–27

Litjens G, Kooi T, Bejnordi BE, Setio AAA, Ciompi F, Ghafoorian M, Van Der Laak JA, Van Ginneken B, Sánchez CI (2017) A survey on deep learning in medical image analysis. Med Image Anal 42:60–88

Liu J, Huang Q, Xia X, Shihab E, Lo D, Li S (2020) Is using deep learning frameworks free? Characterizing technical debt in deep learning frameworks. In: Proceedings of the 42nd ACM/IEEE international conference on software engineering - SE in society (ICSE'20 SEIS), ACM/IEEE

Liu Z, Huang Q, Xia X, Shihab E, Lo D, Li S (2018) Satd detector: a text-mining-based self-admitted technical debt detection tool. In: Proceedings of the 40th international conference on software engineering: companion proceedings, ACM, pp 9–12

Ma L, Juefei-Xu F, Zhang F, Sun J, Xue M, Li B, Chen C, Su T, Li L, Liu Y, et al. (2018) Deepgauge: multi-granularity testing criteria for deep learning systems. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, ACM, pp 120–131

Maldonado E, Shihab E (2015) Detecting and quantifying different types of self-admitted technical debt. In: Proceedings of the 7th IEEE international workshop on managing technical debt (MTD'15), pp 9–15

Maldonado E, Abdalkareem R, Shihab E, Serebrenik A (2017) An empirical study on the removal of self-admitted technical debt. In: Proceedings of the 33rd international conference on software maintenance and evolution (ICSME'17), IEEE

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. Ann Math Stat: 18(1):50–60

Mantel N (1966) Evaluation of survival data and two new rank order statistics arising in its consideration. Cancer Chemother Rep 50:163–170

McElreath R (2020) Statistical rethinking: A Bayesian course with examples in R and Stan. CRC Press

McHugh ML (2012) Interrater reliability: the kappa statistic. Biochemia Medica: Biochemia Medica 22(3):276–282

Miller RG Jr (2011) Survival analysis, vol 66. Wiley

Pei K, Cao Y, Yang J, Jana S (2017) Deepxplore: automated whitebox testing of deep learning systems. In: Proceedings of the 26th symposium on operating systems principles, ACM, pp 1–18

Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: Proceedings of the 30th IEEE international conference on software maintenance and evolution (ICSME'14), pp 91–100

Quesenberry CP Jr, Fireman B, Hiatt RA, Selby JV (1989) A survival analysis of hospitalization among patients with acquired immunodeficiency syndrome. Am J Public Health 79(12):1643–1647

Sallab AE, Abdou M, Perot E, Yogamani S (2017) Deep reinforcement learning framework for autonomous driving. Electronic Imaging 2017(19):70–76

Sculley D, Holt G, Golovin D, Davydov E, Phillips T, Ebner D, Chaudhary V, Young M, Crespo JF, Dennison D (2015) Hidden technical debt in machine learning systems. In: Advances in neural information processing systems, pp 2503–2511

Seaman C, Guo Y (2011) Measuring and monitoring technical debt. In: Advances in Computers, Elsevier, vol 82, pp 25–46

Shalev-Shwartz S, Shammah S, Shashua A (2016) Safe, multi-agent, reinforcement learning for autonomous driving. arXiv:161003295

Shull F (2011) Perfectionists in a world of finite resources. IEEE Softw 28(2):4–6

Spencer D (2009) Card sorting: designing usable categories. Rosenfeld Media

Spínola RO, Vetrò A, Zazworka N, Seaman C, Shull F (2013) Investigating technical debt folklore: shedding some light on technical debt opinion. In: 2013 4th international workshop on managing technical debt (MTD), IEEE, pp 1–7

Sun Y, Huang X, Kroening D (2018a) Testing deep neural networks. arXiv:180304792

Sun Y, Wu M, Ruan W, Huang X, Kwiatkowska M, Kroening D (2018b) Concolic testing for deep neural networks. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, ACM, pp 109–119

Tian Y, Pei K, Jana S, Ray B (2018) Deeptest: automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th international conference on software engineering, ACM, pp 303–314

Wehaibi S, Shihab E, Guerrouj L (2016) Examining the impact of self-admitted technical debt on software quality. In: IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), 2016, IEEE, vol 1, pp 179-188

Wei L, Lachin J (1984) Two-sample asymptotically distribution-free tests for incomplete multivariate observations. J Am Stat Assoc 79(387):653–661

Zampetti F, Serebrenik A, Di Penta M (2018) Was self-admitted technical debt removal a real removal? An in-depth perspective. In: 2018 IEEE/ACM 15th international conference on mining software repositories (MSR), IEEE, pp 526–536

Zampetti F, Serebrenik A, Di Penta M (2020) Automatically learning patterns for self-admitted technical debt removal. In: 2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER), IEEE, pp 355–366

Zazworka N, Shaw MA, Shull F, Seaman C (2011) Investigating the impact of design debt on software quality. In: Proceedings of the 2nd workshop on managing technical debt, ACM, New York, NY, USA, MTD '11, pp 17–23. https://doi.org/10.1145/1985362.1985366

Zazworka N, Spínola RO, Vetro A, Shull F, Seaman C (2013) A case study on effectively identifying technical debt. In: Proceedings of the 17th international conference on evaluation and assessment in software engineering, ACM, New York, NY, USA, EASE '13, pp 42–47. https://doi.org/10.1145/2460999.2461005

Zhang M, Zhang Y, Zhang L, Liu C, Khurshid S (2018a) Deeproad: gan-based metamorphic testing and input validation framework for autonomous driving systems. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, ACM, pp 132–142

Zhang Y, Chen Y, Cheung SC, Xiong Y, Zhang L (2018b) An empirical study on tensorflow program bugs. International Symposium on Software Testing and Analysis

Jiakun Liu is currently a Ph.D. student in the College of Computer Science and Technology, Zhejiang University, China. His research interests include mining software repositories and empirical software Engineering.

Qiao Huang is currently a Ph.D. student in the College of Computer Science and Technology, Zhejiang University, China. He received both of his bachelor and master's degrees in computer science and software engineering from Zhejiang University in 2012 and 2016. His current research interests include mining software repositories and empirical software engineering.

Xin Xia is an ARC DECRA Fellow and a lecturer at the Faculty of Information Technology, Monash University, Australia. Prior to joining Monash University, he was a post-doctoral research fellow in the software practices lab at the University of British Columbia in Canada, and a research assistant professor at Zhejiang University in China. Xin received both of his Ph.D and bachelor degrees in computer science and software engineering from Zhejiang University in 2014 and 2009, respectively. To help developers and testers improve their productivity, his current research focuses on mining and analyzing rich data in software repositories to uncover interesting and actionable information. More information at: https://xin-xia.github.io/.

Emad Shihab is Associate Dean of Research and Graduate Studies and Associate Professor in the Gina Cody School of Engineering and Computer Science at Concordia University. He holds a Concordia University Research Chair in Software Analytics. His research interests are in Software Engineering, Mining Software Repositories, and Software Analytics. Dr. Shihab received the 2019 MSR Early Career Achievement Award and the 2019 CS-CAN/INFO-CAN Outstanding Young Computer Science Researcher Prize. He is recognized as a leader in the field, serving on numerous steering and organization committees of core software engineering conferences. His work has been done in collaboration with world-renowned researchers from Australia, Brazil, China, Europe, Japan, the United Kingdom, Singapore and the USA and adopted by some of the biggest software companies, such as Microsoft, Avaya, BlackBerry, and Ericsson. He is a senior member of the IEEE. His homepage is: http://das.encs.concordia.ca/.

David Lo is a ACM Distinguished Member and an Associate Professor of Information Systems at Singapore Management University. He received his PhD degree in Computer Science from National University of Singapore in 2008. His research interest is in the intersection of software engineering and data science, encompassing socio-technical aspects and analysis of different kinds of software artefacts, with the goal of improving software quality and developer productivity. His work has been published in premier and major conferences and journals in the area of software engineering, AI, and cybersecurity.

Shanping Li received his Ph.D. degree from the College of Computer Science and Technology, Zhejiang University in 1993. He is currently a professor in the College of Computer Science and Technology, Zhejiang University. His research interests include Software Engineering, Distributed Computing, and the Linux Operating System.

Affiliations

Jiakun Liu[1] · Qiao Huang[1] · Xin Xia[2] · Emad Shihab[3] · David Lo[4] · Shanping Li[1]

Jiakun Liu
jkliu@zju.edu.cn

Qiao Huang
tkdsheep@zju.edu.cn

Emad Shihab
eshihab@encs.concordia.ca

David Lo
davidlo@smu.edu.sg

Shanping Li
shan@zju.edu.cn

1 College of Computer Science and Technology, Zhejiang University, Hangzhou, China

2 Faculty of Information Technology, Monash University, Melbourne, Australia

3 Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada

4 School of Information Systems, Singapore Management University, Singapore, Singapore