A First Look at the Lifecycle of DL-Specific Self-Admitted Technical Debt

Gilberto Recupito, Vincenzo De Martino, Dario Di Nucci, Fabio Palomba
SeSa Lab - University of Salerno, Fisciano, Italy
{grecupito, vdemartino, ddinucci, fpalomba}@unisa.it

***Abstract*—The rapid adoption of Deep Learning (DL)-enabled systems has revolutionized software development, driving innovation across various domains. However, these systems also introduce unique challenges, particularly in maintaining software quality and performance. Among these challenges, Self-Admitted Technical Debt (SATD) has emerged as a growing concern, significantly impacting the maintainability and overall quality of ML and DL-enabled systems. Despite its critical implications, the lifecycle of DL-specific SATD—how developers introduce, acknowledge, and address it over time—remains underexplored. This study presents a preliminary analysis of the persistence and lifecycle of DL-specific SATD in DL-enabled systems. The purpose of this project is to uncover the patterns of SATD introduction, recognition, and durability during the development life cycle, providing information on how to manage these issues. Using mining software repository techniques, we examined 40 ML projects, focusing on 185 DL-specific SATD instances. The analysis tracked the introduction and persistence of SATD instances through project commit histories to assess their lifecycle and developer actions. The findings indicate that DL-specific SATD is predominantly introduced during the early and middle stages of project development. Training and Hardware phases showed the longest SATD durations, highlighting critical areas where debt accumulates and persists. Additionally, developers introduce DL-specific SATD more frequently during feature implementation and bug fixes. This study emphasizes the need for targeted DL-specific SATD management strategies in DL-enabled systems to mitigate its impact. By understanding the temporal characteristics and evolution of DL-specific SATD, developers can prioritize interventions at critical stages to improve the maintainability and quality of the system.**

***Index Terms*—Technical Debt, Deep Learning, SE4AI, Empirical Software Engineering**

I. Introduction

Deep Learning (DL) has rapidly established itself as a transformative force across diverse industries. By leveraging advanced neural network architectures, DL provides powerful solutions to complex challenges, revolutionizing fields such as healthcare, autonomous driving, and natural language processing [1]–[3]. DL is a subfield of Machine Learning (ML), a broader field focused on developing algorithms that learn from data [4], [5]. ML techniques, including DL, are increasingly integrated into traditional software systems, resulting in ML-enabled systems that incorporate at least one ML component [6]. Although the adoption of these technologies in software systems has spread significantly to provide novel services, they present unique challenges that can increase the effort required to maintain and ensure the high quality of the architecture of ML-enabled systems [7]. One of the main challenges that can negatively impact the quality of software systems, particularly DL-enabled systems, is related to technical debt [8]. This term refers to the issues arising when developers opt for quick fixes or suboptimal solutions during the development of machine learning models [9], [10]. Over time, accumulating technical debt can lead to significant quality degradation, affecting not only maintainability but also other aspects of DL-specific quality such as performance and system security [11].

Developers acknowledge the presence of these quality issues explicitly, often through Self-Admitted Technical Debt (SATD) [12], which highlights potential problems in source code comments or issue trackers [13]. Various methods have been proposed to help developers manage SATD, noting that comments usually include recognizable keywords such as *'TODO'* and *'FIXME'* [14]. An example of introducing an SATD is shown in the Listing I. Developers add comments to raise an issue and to describe the suboptimal solution they employed, using the keyword (i.e., "TODO") to warn of the need for further rework in the future, establishing a debt.

```

def get_dummy_inputs(self, device, seed=0):
# TODO: use tensor inputs instead of PIL, this is
    here just to leave the old expected_slices
    untouched
    image = floats_tensor((1, 3, 32, 32), rng=
    random.Random(seed)).to(device)@@
    image=image.cpu().permute(0, 2, 3, 1)[0]
    init_image = Image.fromarray(np.uint8(image)).
    convert("RGB").resize((64, 64))
    mask_image = Image.fromarray(np.uint8(image +
    4)).convert("RGB").resize((64, 64))
```

Listing 1: Example of the introduction of a SATD.

Therefore, DL-specific SATD presents unique challenges compared to conventional SATD because of the distinct characteristics of DL components. Developers must consider SATD from conventional systems and those specific to ML and DL software, which are particularly vulnerable to the accumulation of technical debt [9]. One notable example is the work conducted by Pepe et al. [15], which defined a taxonomy of 41 distinct types of DL-specific SATD specific to DL-enabled systems. These types of SATD encompass various phases of DL, including data, model, training, and inference.

While previous efforts have defined DL-specific SATD in ML projects by creating a taxonomy, they have not thoroughly examined the life cycle of these technical debts. Examining when SATD instances are introduced and determining how long they persist provides critical information on the dynamics of SATD management. Furthermore, by investigating the longevity and timing of SATD instances, we can better understand their impact on software quality, maintenance efforts, and the overall lifecycle of DL-enabled systems.

Based on this gap, this paper presents a preliminary mining repository study to investigate the life cycle of DL-specific SATD in order to understand (1) when and for how long developers acknowledge the DL-specific SATD and (2) what tasks developers perform when they acknowledge the presence of DL-specific SATD in DL-enabled systems. To achieve this goal, we build on top of the work done by Pepe et al. [15] by using 41 distinct types of DL-specific SATD and 40 Python open-source projects relying on TensorFlow or PyTorch. The study reveals that DL-specific SATD is predominantly introduced during the early and middle development stages, with minimal instances acknowledged in the later stages. Additionally, the lifespan of SATD varies significantly across different components. SATD related to *Hardware* and *Training* phases tend to last longer, while those associated with *API* and *Pipeline* phases are typically resolved more quickly. Specifically, most SATD were identified during feature implementation (64 instances) and bug-fixing tasks (55 instances), underscoring the prevalence of technical debt in these development activities. These insights highlight the nuanced dynamics of DL-specific SATD, emphasizing the need for tailored strategies to manage and mitigate effectively. This preliminary work is important for software engineering researchers and developers who want to understand the life cycle of DL-specific SATD and the activities performed when the SATD is recognized in DL-enabled systems, highlighting the need for tools and models to capture, track, and address DL-specific SATD across development phases and components, enabling targeted interventions. This study highlights the importance of SATD management during feature implementation and fixes, where these debts are most common. Additionally, variations in SATD duration across components underline the need for customized solutions, such as automated tools and long-term strategies to resolve technical debts.

II. Related Work

The management of SATD in DL-enabled systems is increasingly crucial due to the inherent complexities and distinctive challenges these systems present. OBrien et al. [16] investigate SATD in ML-enabled systems by analyzing 68,820 SATD comments from 2,641 Python-based ML repositories. Their study proposes a taxonomy of 23 ML-specific SATD types, focusing on qualitative aspects such as awareness, modularity, readability, and performance, mainly covering shallow ML projects. Liu et al. [17] analyze SATD in seven popular DL frameworks, identifying 7,159 SATD comments through manual classification and categorizing them into seven types: design, defect, documentation, requirement, test, compatibility, and algorithm debts, with design debt being the most prevalent. Building on this work, Liu et al. [18] further explore the introduction and removal patterns of these debt types, finding that design debt is not only the most common but also the fastest to be resolved, offering deeper insights into its lifecycle. Bhatia et al. [8] conducted a study investigating SATD in 318 Python ML projects compared to 318 non-ML Python projects. Their findings revealed that ML projects exhibit 2.1 times more SATD, primarily due to frequent code replacements and experimental development. This research expands the existing SATD taxonomies to incorporate specific types related to ML, maps SATD occurrences to various stages of the ML pipeline, and employs survival analysis to examine the dynamics of SATD over time. While previous research focuses on shallow ML, Pepe et al. [15] created a taxonomy of 41 distinct types of DL-specific SATD analyzing 40 DL-projects based on two popular frameworks, Pytorch and TensorFlow.

Acknowledging the work done in the literature to improve knowledge about ML/DL-specific SATD, our work advances the technical debt field by analyzing when and how developers recognize SATD and the tasks performed when developers acknowledge SATD presence. This provides insight into its evolution through different stages of development and components. Unlike previous studies that mainly classify SATD types in traditional or ML-enabled systems, our research focuses on DL-specific SATD. This work contributes new perspectives on the interaction between SATD, development practices, and system quality in advanced DL pipelines.

III. Research Questions

To investigate when and how developers acknowledge the presence of SATD, our study adopts the Goal-Question-Metric (GQM) approach [19].

> ***Analyze*** *the lifecycle of DL-specific SATD* ***for the purpose of*** *investigating when and how SATD is acknowledged* ***from the point of view*** *of developer and researchers* ***in the context of*** *DL-enabled system development and maintenance.*

In particular, researchers are interested in the life cycle of DL-specific SATD to develop approaches for identifying, analyzing, and mitigating DL-specific SATD. Developers are interested in the analysis of the SATD to understand how to reduce risk and maintain system quality. To achieve our research goal, we focus on two main research questions **(RQs)** that guide our study. The **(RQ_1)**, seeks to identify when developers most frequently signal SATD during the software lifecycle (e.g., hardware, data, or training) and how long these issues remain unresolved. Understanding when SATD were first recognized provides insight into developers' behavior in managing technical debt and their interest in resolving it. In particular, we ask:

> **RQ_1:** *When and for how long do developers acknowledge DL-specific SATD in DL-enabled systems?*

TABLE I: Definition of SATD categories.

Category	Description	Example
Hardware	This category accounts for SATD dealing with hardware components, e.g., GPUs or TPUs, used for training the DL model or during inference once the trained model is deployed in a production environment.	`TODO: do something with loras and offloading to CPU`
API	This category accounts for SATD due to the usage and integration of DL frameworks.	`TODO: support native HF versions of MusicGen.`
Data	This category groups SATD related to the format/shape of the data used to train a model.	`` TODO: maybe have a cleaner way to cast the input (from `ImageProcessor` side?) ``
Model	This category includes SATD concerning the design and setting of the DL model.	`TODO: attention mask is not used`
Training	This category encompasses SATD related to the training process, including the selection of loss functions, initialization of model parameters, learning strategies, and suboptimal implementation of training logic.	`TODO: Retrieve the seeds from the model definition instead.`
Inference	This category includes SATD related to inappropriate post-processing of the outcome generated when running the trained model, as well as using suboptimal prompting.	`TODO: add prompts to .yaml`
Pipeline	This category groups SATD dealing with the setting of the DL pipeline in terms of its design and optimization.	`TODO: enable higher-order gradients`

After assessing the lifecycle of DL-specific SATD, the (**RQ$_2$**) aims to identify the specific actions that developers take when they recognize DL-specific SATD. These insights provide an understanding of whether SATD arises during feature development, bug fixing, or other activities.

> **RQ$_2$:** *What tasks do developers perform when acknowledging the presence of DL-specific SATD in DL-enabled systems?*

To design our process and analyze the results of our mining study, we followed the guidelines specified in the "Repository Mining" and "General Standard" categories of the ACM/SIGSOFT Empirical Standards [20].

IV. Research Method

This section describes our study research approach to answer our (**RQs**).

A. Dataset Description

The first step is to select appropriate projects within DL-specific SATD for our analysis. For this purpose, we built our study upon the work of Pepe et al. [15] for a twofold reason.

On the one hand, the contribution of Pepe et al. [15] offers a curated dataset of 100 ML open-source projects, with 46 affected by SATD, providing a strong basis for our study. These projects are carefully selected to ensure relevance, focusing on Python-based repositories utilizing the two well-known DL frameworks, PyTorch[1] and TensorFlow[2]. This dataset was refined by excluding tutorials, toy projects, and repositories not primarily written in Python, ensuring the inclusion of active, high-quality, and representative projects. Moreover, the dataset guarantees that all projects included contain at least one DL-specific SATD.

On the other hand, Pepe et al. [15] contribute a comprehensive taxonomy of DL-specific SATD that classifies and contextualizes the technical debts found in ML projects. This taxonomy, in Table I, classifies 200 SATD instances into 41 distinct types specific to DL. The types are further grouped under seven main categories, referring to aspects such as hardware configurations, training processes, data handling, and inference issues, all of which are essential in the lifecycle of DL-enabled systems.

To maintain the relevance of the SATD instances in the dataset, we conducted a verification process to confirm the data's reliability and validity. Specifically, we verified whether the GitHub repositories associated with the identified SATD instances were still accessible and actively maintained at the time of our study. Specifically, for each of the SATD instances analyzed with the validated dataset, we relied on Github API [21] to ensure the existence of the repository and the related commit for which the SATD has been identified. This step guarantees that all the SATD selected as the starting set of experimental objects are available to extract the commit history. Therefore, after this step, we selected a total of 185 SATD instances across 40 GitHub projects. By focusing on this refined subset, we ensured that the SATD instances and their corresponding projects were relevant to our study's objective of analyzing technical debt in DL-enabled systems.

B. History Mining

After selecting the project dataset and the catalog of DL-specific SATD, we analyzed the commits in Github projects. In particular, we proceeded to the commit extraction phase to identify the specific commits where each DL-specific SATD was first introduced into the codebase. To achieve this, we uti-

[1] https://pytorch.org/
[2] https://www.tensorflow.org

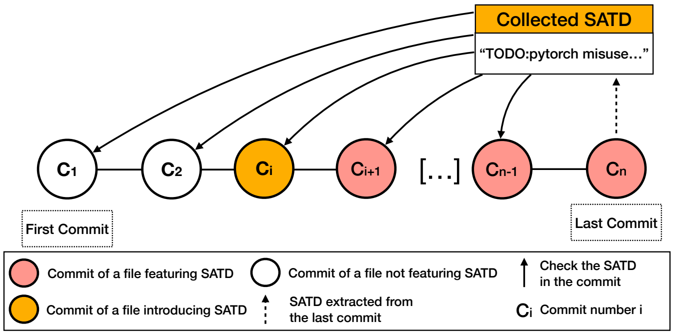

Fig. 1: SATD introducing commit extraction

lized PyDriller[3], a Python framework designed for mining Git repositories and analyzing commit histories. PyDriller allowed us to programmatically retrieve detailed information for each commit, including the commit hash, the lines modified, and the associated commit message. This simplified the data extraction process and ensured a systematic approach to managing the large volume of commits in projects. To determine the specific commit that introduced each DL-specific SATD instance, we employed a reverse tracing strategy similar to that performed by Tufano et al. [22], commonly used for analyzing the history of code smells. As depicted in Figure 1, the process began by leveraging the last commit to extract the SATD instance and to trace backward through the commit history. Leveraging the DL-specific SATD annotations, we systematically traversed the commit history in reverse chronological order, examining earlier commits to locate the exact introduction point of the DL-specific SATD instance. This backward traversal method ensured accurate mapping between DL-specific SATD instances and their origin commits.

When analyzing the commit history of a project P. Let a file F in a project P identify the introduction of a specific $s \in$ DL-specific SATD. Two scenarios can arise when encountering a commit that modifies the F file and the prior version of F does not include the specified DL-specific SATD:

1) **Last occurrence but not the first:** This commit marks the most recent occurrence of the $s \in$ DL-specific SATD in the file. However, it does not necessarily represent the first introduction of the DL-specific SATD, as it might have been added earlier in the commit history.
2) **First occurrence:** If no earlier versions of the F file, as examined through the commit history, contain the $s \in$ DL-specific SATD, then this commit is identified as the one that introduces s. In this case, it is considered the introducing commit for the DL-specific SATD.

This approach ensures that all occurrences of $s \in$ DL-specific SATD, whether it is the first occurrence or not, are accurately recorded and analyzed. Specifically, we defined the introduction commit c_i for the $s \in$ DL-specific SATD as the first commit in which the DL-specific SATD annotation appeared. Using this approach, we identified commits that introduce SATD across all file versions in the commit history, collecting a total of 5,337 commits.

[3]https://github.com/ishepard/pydriller

C. Data Analysis

The following section describes our approach to analyzing the data collected to answer our research questions. To address (**RQ**$_1$), we conducted a comprehensive analysis focusing on temporal data of the commit history associated with DL-specific SATD. Specifically, we developed a Python script that writes this information into a `csv` file. Once a $s \in$ DL-specific SATD is found in an F File, s is saved, and the commit history of F is traversed backward until the introduction of s. Additionally, in the saved dataset, we extract the current commit date and the previous commits, collecting the whole history for each file featuring the presence of a DL-specific SATD instance. Moreover, through the backward tracing of the commit history, we record the line number for every commit affected by a specific SATD instance to control the presence of the SATD also when the file changes along the history.

When SATD is introduced. To determine the introduction of any s, we measured the time and number of commits from the start of the project to the most recent commit, where it was still present. To analyze the introduction period of DL-specific SATD in their project lifetime, we examined their position in the commit history. Specifically, we calculated the difference in the number of commits between the repository's first commit and the commit that introduced DL-specific SATD. This analysis allows us to identify at what point in the project developers had reported s, measured by the number of commits made prior to its introduction. Collecting all SATD introduction positions over the project lifetime, we segmented the commit history of each project into three phases:

- **Early Stage**: The first 25% of commits in the project timeline.
- **Middle Stage**: Commits between 25% and 75% of the project timeline.
- **Last Stage**: The final 25% of commits in the project timeline.

This segmentation provided a perspective that highlights overarching insights on when DL-specific SATD-introduction commits occur within a project's lifespan.

How long SATD persists. Additionally, to examine the persistence of DL-specific SATD instances, we evaluated how long they remained unresolved by calculating the number of days between their introduction and the latest commit in the repository. It is important to note that the DL-specific SATD dataset used in this study includes only DL-specific SATD instances that are still present in the project's codebase. These instances were identified from the last active commit and traced backward through the commit history.

This analysis made it possible to determine how long developers continued to work on the project while leaving DL-specific SATD instances unresolved. Indicating the length of time DL-specific SATD instances were present in the system until the latest project version.

Combining the previous two analyses allows us to obtain information on commits that introduce SATD over the project lifetime, highlighting when they are introduced and how long they persist, answering to **RQ**$_1$.

To address (**RQ**$_2$), we focused on examining the commit messages associated with the introducing commits. Commit messages were analyzed using keyword-matching strategies to identify activities in which developers acknowledged DL-specific SATD. Based on the presence of keywords, the analysis categorized code modifications into four main types: feature development, bug fixes, enhancement, and refactoring.

The selection of these types has been found to be effective in classifying commit messages [22]. By using keyword-matching strategies, we systematically categorize SATD acknowledgment specific to DL in commit messages and analyze the activities that lead developers to indicate technical debt. Although this strategy helped us identify some categories, it did not allow for the identification of all categories. In detail, starting from a set of 185 DL-specific SATD commits, we extracted 62 commits under the four starting categories through keyword matching.

Therefore, the first two authors manually analyzed the remaining set of 123 commits by conducting content analysis [23]. Every inspector reviewed every comment, trying to classify it into one of the categories predefined previously. In case of disagreement, the two inspectors analyzed the results and started a discussion to reach an agreement. After this phase, 108 commits were manually categorized into the categories of Table I, still leaving 15 cases of commits introducing DL-specific SATD uncategorized. Finally, a new round of inspection on the remaining part has been done to identify new categories among the meaningful commits. Therefore, a new category, named ”chore”, has been identified in addition.

Specifically, 11 out of the 15 commits are grouped into this category, identifying commit operations that aim to perform maintenance tasks that do not directly affect the software's functionality or features, such as updating dependencies or improving documentation.

Therefore, we analyzed the following categories:

1) **Bug fixes:** where DL-specific SATD were acknowledged during defect resolution or maintenance tasks, identified by keywords such as *fix* or *bug*;
2) **Enhancement:** where DL-specific SATD were acknowledged during the improvement of existing features or functionalities, often identified by keywords such as *improve* or *enhance*;
3) **Feature development:** where DL-specific SATD were introduced alongside the implementation of new functionalities, identified by keywords such as *feature* or *add*;
4) **Refactoring tasks:** where DL-specific SATD were tied to incomplete or deferred structural changes, often marked by keywords like *refactor*.
5) **Chore:** where DL-specific SATD were tied to routine maintenance, such as dependency updates or configuration changes, identified manually.

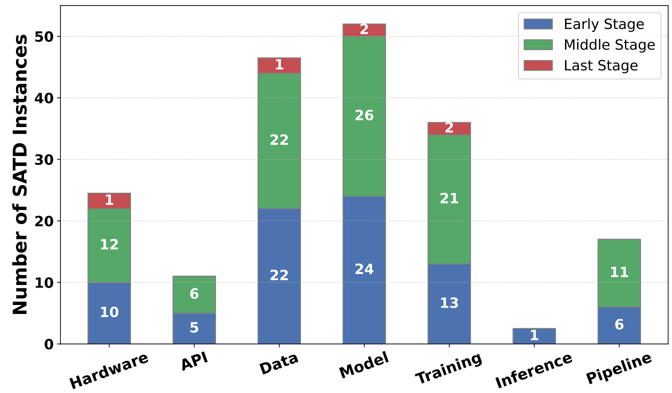

Fig. 2: Frequency of SATD for each development stage.

Four commits introducing DL-specific SATD were not categorized and had no meaningful description; to avoid possible misclassification, we excluded these commits from our analysis. Finally, we classified 181 SATD-introducing commits into their specific five categories. Collecting the occurrences of commit categories that introduce allows us to investigate what are the activities that lead developers to recognize and denote DL-specific SATD, answering to **RQ**$_2$. The entire process to reproduce our work is in the online appendix [24].

V. Results

In this section, we report quantitative insights from the repository mining study to address the **RQs**:

A. RQ$_1$: When and for how long do developers acknowledge DL-specific SATD in DL-enabled systems?

Figure 2 presents the boxplot of the distribution of DL-specific SATD instances across the early, middle, and last stages of development for various components within DL-enabled systems across 40 projects. The analysis reveals that DL-specific SATD is most frequently acknowledged during the middle stage (98), followed by the early stage (81), and only minimally in the last stage (6). This trend underscores that developers primarily recognize and address technical debt early and during the refinement phases of the lifecycle.

Considering the totality of all projects, the *Model* phase has the highest total D-specific SATD instances (52), with 26 cases in the middle stage and 24 instances in the early. Only two are in the last stage. The *Data* phase is the second most frequent (45), with 24 in the middle stage, 22 instances in the early stage, and only one in the last stage.

The *Training* phase follows, with a total of 36 instances, distributed in the middle stage with 21 instances, while the early stage accounts for 13 instances, and only 2 are acknowledged in the last stage. The *Pipeline* phase shows 17 instances in total, with 11 occurring in the early stage and 6 in the middle stage, with no debt reported in the last stage. In contrast, the *Hardware* phase accounts for 22 instances, mainly concentrated in the middle stage (12 instances) and the early stage (10 instances), with none reported in the last stage.

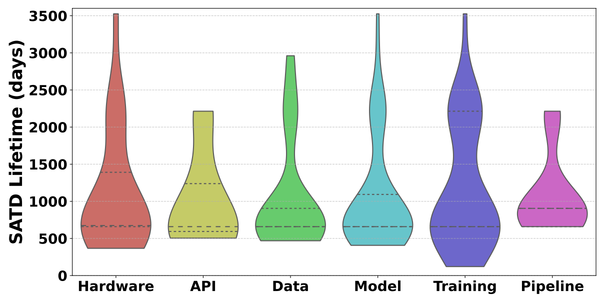

Fig. 3: Distribution of SATD Lifetime since their introduction.

Category	B	F	E	C	R	Total
Hardware	7	5	**8**	1	1	22
API	3	**4**	2	0	1	10
Data	**16**	15	8	5	0	44
Model	14	**20**	14	1	2	51
Training	6	**15**	6	2	7	36
Inference	0	**1**	0	0	0	1
Pipeline	**5**	4	3	2	3	17
Total	55	**64**	41	9	12	181

TABLE II: Distribution of SATD Types Across Categories. Bug Fix (B), Feat (F), Enhance (E), Chore (C), Refactor (R)

The *API* phase has the lowest number of instances among the major phases, with a total of 11 cases: 5 in the early stage, 6 in the middle stage, and none in the last stage. Finally, the *Inference* phase shows only one instance in the early stage.

The findings reveal a trend where most DL-specific SATD is reported in the early and middle stages, with only residual debt persisting into the last stage across most components.

The violin plot shows in Figure 3 the distribution of SATD lifetimes, measured in days, across various components of the DL-enabled system. The diagram reveals the variability in the persistence of DL-specific SATD across components. For example, *Hardware* and *Training* phases exhibit the broadest distributions, with some instances persisting for extended periods, indicating that technical debt in these areas can remain unresolved for years. The median lifetimes for these components suggest that resolving such debt is often delayed, likely due to the complexity or lower prioritization of these issues. In contrast, components such as *Pipeline* and *API* phases exhibit a constrained distribution with shorter lifetimes. This indicates that DL-specific SATD in these areas is typically addressed more promptly and consistently. Meanwhile, *Model* and *Data* phases fall somewhere in between, with DL-specific SATD lifetimes showing moderate distributions and occasional instances of long-lasting debt. This reflects a mix of challenges that are resolved at varying rates. Overall, the plot highlights the differing persistence of DL-specific SATD across components, emphasizing the need for targeted strategies to tackle long-lived technical debt.

☞ **Answer to RQ$_1$.** Developers mainly identify DL-specific SATD during the early (81 instances) and middle (98 instances) stages, with few instances (6) in the final stage. SATD is most common in the *Model* and *Data* and less in the *API* and *Inference* stages. The duration of SATD varies significantly, with the *Hardware* and *Training* phases having the longest-lasting issues. In contrast, the *Pipeline* and *API* phases are resolved more quickly. Generally, developers address most SATD by the refinement phase, although some problems persist longer in certain components.

B. RQ$_2$: What tasks do developers perform when acknowledging the presence of DL-specific SATD in DL-enabled systems?

Table II reports the occurrences of the commit messages that introduce the DL-specific SATD, grouped in the seven categories. Analyzing the commit messages, we found a high occurrence (64 instances) of DL-specific SATD introduced during the *implementation of a new feature* operation. Still, many DL-specific SATD are consequently introduced during a *bug-fixing* operation (55 instances) and operations related to the *enhancement* of the system (41 instances). Conversely, a lower amount of instances are introduced during a *refactoring* (12 instances) or a *chore* (9 instances).

DL-specific SATD is almost equally present across different operations categories. However, the analysis varies significantly when considering the specific types of DL-related SATD. From the set of commit messages analyzed, we found that 8 of the 22 *Hardware* related DL-specific SATD are introduced during an enhancement operation, resulting in the most recurrent in this category. Data-related DL-specific SATD usually appears when a *bug-fixing operations appear*. For instance, in the project *huggingface/diffusers*, during a bug-fixing operation related to the feature extraction component, a data-related SATD has been denoted by the developer's raising the absence of a scaling operation [25].

The most recurrent operation that introduces Model-related SATD is the implementation of the new feature with 20 instances. From the same commit operation related to the release of a new version of structured prompting, retrieved in the project *microsoft/LMOps*, we found the introduction of the developers of four Model-related SATD, raising problems related to the encoding components of transformers-based models (e.g., "*# TODO: we should reuse the pretrained model dict which already has mask*") [26].

Training-related SATD is mostly introduced during the implementation of a new feature, and 15 instances were found for this operation. Also, in this case, the addition of a new feature is denoted by developers to cause problems in the components that are involved in the training phase. Specifically, a developer denoted the introduction of a dummy encoder for the Roberta model to fix the non-determinism model (i.e., "*# TODO: remove after fixing the non-deterministic text encoder*") when introducing a new component in the system [27]. Among the 17 instances of pipeline-SATD, we found that every commit-operation type introduces at least one SATD of this category, with a slightly higher tendency when performing bug-fixing operations, with 5 instances. Finally, the unique SATD related to the inference of DL models was found to be introduced during the implementation of a new feature.

In summary, there is a higher tendency to highlight DL-specific SATD when a new feature is implemented. Moreover, four of the categories are found to be introduced during this type of commit operation. At the same time, *Pipeline*, *Hardware*, and *API* SATD instances are occurrences that are distributed among different categories of commit operations.

☞ **Answer to RQ$_2$.** Developers more frequently introduce DL-specific SATD during feature implementation (64) and bug fixes (55). In contrast, SATD is less commonly associated with refactoring (12) or chore categories (9). DL-specific SATD introduced also varies based on the activity: *Data* SATD is often encountered during bug fixes (16), while *Model* and *Training* SATD are typically introduced during the development of new features (20 and 15 respectively).

VI. Threats to Validity

This section outlines the potential limitations and our strategies for addressing them.

External Validity. One of the main challenges is the representativeness of the data sample used in our investigation of DL-specific SATD. To mitigate the threat to generalizability of the dataset, we relied on the study of Pepe et al. [15] consisting of 40 open-source Python projects using TensorFlow and PyTorch and 185 DL-specific SATD types and applied a verification process to confirm the validity of the data. Although this sample provides a basis for our investigation, it may not reflect the diversity of DL-specific SATD types across software systems and contexts, since it does not include resolved SATD instances. Therefore, replications are needed to evaluate our preliminary observations, expanding the study to a larger scale. Additional investigations allow for the identification of additional DL-specific SATD types and the discovery of patterns among them. An additional threat concerns the diversity of the ML Projects affected by DL-specific SATD. To mitigate this threat, the Pepe et al. [15] dataset contains only projects that have at least one SATD instance, ensuring the reliability of the dataset. Additionally, our study focuses on PyTorch and Tensorflow projects, given its widespread use.

Construct Validity. One potential threat arises from the classification of DL-specific SATD instances based on keyword matching and manual inspection. Misinterpretation of SATD categories or commit messages could lead to incorrect classifications. To address this, we leverage the taxonomy of DL-specific SATD [15] to ensure the use of consistent definitions, and we conducted content analysis sessions [23] to minimize bias in the manual categorization. Disagreements between inspectors were resolved through discussions to reach consensus. The data analysis was conducted by the first two authors, who have substantial expertise in AI software engineering. The first author is a Ph.D. student with over four years of experience in tech debt and software engineering for artificial intelligence (SE4AI). The second author is a Ph.D. student with over three years of experience in DL and SE4AI.

VII. Discussion

Analysis of the prolonged persistence of DL-specific SATD revealed several discussion points.

The need for the prevention of SATD. Our results indicate that developers frequently recognize DL-specific SATD during the initial and middle stages of DL projects across all categories analyzed. Notably, 64 DL-specific SATD instances are introduced during feature implementation, suggesting that these issues often emerge during active development when developers prioritize meeting deadlines or delivering functionality over addressing technical debt. The long persistence of these issues throughout the project lifecycle highlights a tendency to leave acknowledged SATD unresolved. This behavior can lead to compounding challenges, such as reduced system maintainability and potential performance degradation. Addressing SATD early, especially during feature development, is crucial to mitigating these risks and ensuring the long-term sustainability of DL-enabled systems [9], [10].

While DL-specific SATD exhibits unique characteristics, such as challenges in data dependencies, model interpretability, and retraining, our findings suggest similarities with traditional SATD in its upward trend and persistence [28]. This implies that established techniques like code reviews or static analysis could be adapted, although further research is needed to tailor them for DL-specific challenges. Persistent DL-specific SATD can also have broader implications, including risks to system reliability, especially in critical domains.

Combining the complexity of DL with SATD. Although this analysis focuses on the timing of SATD introduction and does not delve into resolution strategies, the long persistence found in these projects has some implications. Analyzing SATD in traditional software systems, Potdar and Shihab [12] discovered that even after many releases, only less than 64% percent of SATD instances are resolved. On the same line, Bavota and Russo [28] also analyzed the lifespan of traditional SATD instances, discovering that these self-acknowledged issues remained unresolved for more than 1000 commits since the start of the project. In the context of DL technologies, we found that many of the DL-specific SATD instances considered in this study, especially for the *training-related* category, remained unresolved for longer periods. Thus, the complexity of managing multiple components, such as pipelines, model architectures, and training processes, combined with the difficulty of dealing with SATD in traditional software systems, suggests that developers face significant hurdles in solving problems that may propagate not only on the current state of the system but also on future iterations within specific DL-systems. This highlights the need for new practices to address the unique challenges faced in DL projects. Without such strategies, the accumulation of unresolved SATDs could intensify the inherent complexity of these systems, threatening their maintainability over time. Therefore, to support developers in managing SATD, there is a pressing need to develop tools tailored to DL-specific contexts. These could include automated SATD tracking integrated into ML pipelines, enhanced visualization of debt accumulation within lifecycle stages, and prioritization mechanisms that quantify technical and operational risks associated with unresolved SATD.

VIII. Conclusion and Future Work

In this study, we analyzed the occurrence and evolution of DL-specific SATD over time in DL-enabled systems to better understand when and how developers acknowledge these quality issues. By analyzing 40 open-source projects and 185 DL-specific instances, our investigation reveals critical insights into the lifecycle of DL-specific SATD. Notably, we observed that DL-specific SATD is mainly acknowledged during the early and middle stages of the DL software lifecycle, often persisting throughout the project's development. Our analysis also revealed a high prevalence of DL-specific SATD instances introduced during the implementation of new features, indicating that DL-specific SATD often emerges when new production code is added to the system.

These two outcomes provide a foundation for investigating DL-specific SATD to determine if and how its persistence after adding new system components leads to quality issues, harming the maintenance process. Therefore, future work will focus on further investigations to evaluate the practical motivations of DL-specific SATD persistence from the developer's perception. Moreover, we aim to expand the dataset to include a larger set of instances, enabling their identification that was acknowledged in the past and subsequently resolved. This extended analysis will facilitate the discovery of common patterns and trends, paving the way for the development of refactoring techniques for DL-specific SATD.

Acknowledgments

This work has been partially supported by the European Union - NextGenerationEU through the Italian Ministry of University and Research, Projects PRIN 2022 "QualAI: Continuous Quality Improvement of AI-based Systems" (grant n. 2022B3BP5S, CUP: H53D23003510006).

References

[1] A. Ahmad, A. Tariq, H. K. Hussain, and A. Y. Gill, "Revolutionizing healthcare: How deep learning is poised to change the landscape of medical diagnosis and treatment," *Journal of Computer Networks, Architecture and High Performance Computing*, 2023.

[2] M. Fisher, "Companies Using Machine Learning: Transforming Industries with AI Solutions — modelsai.org," https://modelsai.org/blog/companies-using-machine-learning, [Accessed 10-11-2024].

[3] P. S. Chib and P. Singh, "Recent advancements in end-to-end autonomous driving using deep learning: A survey," *IEEE Transactions on Intelligent Vehicles*, 2023.

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[5] I. Goodfellow, "Deep learning," 2016.

[6] S. Martínez-Fernández, J. Bogner, X. Franch, M. Oriol, J. Siebert, A. Trendowicz, A. M. Vollmer, and S. Wagner, "Software engineering for ai-based systems: a survey," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–59, 2022.

[7] R. Nazir, A. Bucaioni, and P. Pelliccione, "Architecting ml-enabled systems: Challenges, best practices, and design decisions," *Journal of Systems and Software*, vol. 207, p. 111860, 2024.

[8] A. Bhatia, F. Khomh, B. Adams, and A. E. Hassan, "An empirical study of self-admitted technical debt in machine learning software," *arXiv preprint arXiv:2311.12019*, 2023.

[9] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," *Advances in neural information processing systems*, vol. 28, 2015.

[10] S. McGuire, E. Schultz, B. Ayoola, and P. Ralph, "Sustainability is stratified: Toward a better theory of sustainable software engineering," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1996–2008.

[11] G. Recupito, F. Pecorelli, G. Catolino, V. Lenarduzzi, D. Taibi, D. Di Nucci, and F. Palomba, "Technical debt in ai-enabled systems: On the prevalence, severity, impact, and management strategies for code and architecture," *Journal of Systems and Software*, vol. 216, 2024.

[12] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 91–100.

[13] Y. Li, M. Soliman, and P. Avgeriou, "Automatic identification of self-admitted technical debt from four different sources," *Empirical Software Engineering*, vol. 28, no. 3, p. 65, 2023.

[14] A. Mastropaolo, M. Di Penta, and G. Bavota, "Towards automatically addressing self-admitted technical debt: How far are we?" in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 585–597.

[15] F. Pepe, F. Zampetti, A. Mastropaolo, G. Bavota, and M. Di Penta, "A taxonomy of self-admitted technical debt in deep learning systems," *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2024.

[16] D. OBrien, S. Biswas, S. Imtiaz, R. Abdalkareem, E. Shihab, and H. Rajan, "23 shades of self-admitted technical debt: An empirical study on machine learning software," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 734–746.

[17] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "Is using deep learning frameworks free? characterizing technical debt in deep learning frameworks," in *ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society*, 2020.

[18] ——, "An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks," *Empirical Software Engineering*, vol. 26, pp. 1–36, 2021.

[19] V. R. B. G. Caldiera and H. D. Rombach, "The goal question metric approach," *Encyclopedia of software engineering*, pp. 528–532, 1994.

[20] P. Ralph, N. b. Ali, S. Baltes, D. Bianculli, J. Diaz, Y. Dittrich, N. Ernst, M. Felderer, R. Feldt, A. Filieri *et al.*, "Empirical standards for software engineering research," *arXiv preprint arXiv:2010.03525*, 2020.

[21] "GitHub REST API documentation - GitHub Docs — docs.github.com," https://docs.github.com/en/rest, 2022.

[22] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 403–414.

[23] K. Krippendorff, *Content analysis: An introduction to its methodology*. Sage publications, 2018.

[24] A. Anonymous, "A First Look at the Lifecycle of DL-Specific Self-Admitted Technical Debt - Replication package," 2 2024. [Online]. Available: https://figshare.com/s/5d50b381dfd29719bab0

[25] "Example of Data-related SATD introduced during bug-fixing in huggingface-diffusers project," https://github.com/huggingface/diffusers/blob/3dc97bd1482fb099aa41a15aae55ff45c8f2b042/examples/community/pipeline_zero1to3.py#L667, accessed: 2024-10-30.

[26] "Example of Model-related SATD introduced during the implementation of a new feature in microsoft-LMOps project," https://github.com/microsoft/LMOps/blob/00b5846b6e9c20b088c39fa50412fac46f7bd056/structured_prompting/fairseq-version/fairseq/fairseq/tasks/online_backtranslation.py#L641, accessed: 2024-10-30.

[27] "Example of Training-related SATD introduced during the implementation of a new feature in huggingface-diffusers project," https://github.com/huggingface/diffusers/blob/ba352aea29df9a7f086bf0815fe9fe479218f801/tests/pipelines/altdiffusion/test_alt_diffusion.py#L187, accessed: 2024-10-30.

[28] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proceedings of the 13th international conference on mining software repositories*, 2016, pp. 315–326.