8-20-2024

Challenges and Practices of Deep Learning Model Reengineering: A Case Study on Computer Vision

Wenxin Jiang
Purdue University, jiang784@purdue.edu

Vishnu Banna
Purdue University

Naveen Vivek
Purdue University

Abhinav Goel
Purdue University

Nicholas Synovic
Loyola University Chicago, nsynovic@luc.edu

See next page for additional authors

Empirical Software Engineering (2024) 29:142
https://doi.org/10.1007/s10664-024-10521-0

Challenges and practices of deep learning model reengineering: A case study on computer vision

Wenxin Jiang[1] · **Vishnu Banna**[1] · **Naveen Vivek**[1] · **Abhinav Goel**[1] · **Nicholas Synovic**[2] · **George K. Thiruvathukal**[2] · **James C. Davis**[1]

Accepted: 25 June 2024

Abstract

Context Many engineering organizations are reimplementing and extending deep neural networks from the research community. We describe this process as deep learning model reengineering. Deep learning model reengineering — reusing, replicating, adapting, and enhancing state-of-the-art deep learning approaches — is challenging for reasons including under-documented reference models, changing requirements, and the cost of implementation and testing.

Objective Prior work has characterized the challenges of deep learning model development, but as yet we know little about the deep learning model reengineering process and its common challenges. Prior work has examined DL systems from a "product" view, examining defects from projects regardless of the engineers' purpose. Our study is focused on reengineering activities from a "process" view, and focuses on engineers specifically engaged in the reengineering process.

Method Our goal is to understand the characteristics and challenges of deep learning model reengineering. We conducted a mixed-methods case study of this phenomenon, focusing on the context of computer vision. Our results draw from two data sources: defects reported in open-source reeengineering projects, and interviews conducted with practitioners and the leaders of a reengineering team. From the defect data source, we analyzed 348 defects from 27 open-source deep learning projects. Meanwhile, our reengineering team replicated 7 deep learning models over two years; we interviewed 2 open-source contributors, 4 practitioners, and 6 reengineering team leaders to understand their experiences.

Results Our results describe how deep learning-based computer vision techniques are reengineered, quantitatively analyze the distribution of defects in this process, and qualitatively discuss challenges and practices. We found that most defects (58%) are reported by re-users, and that reproducibility-related defects tend to be discovered during training (68% of them are). Our analysis shows that most environment defects (88%) are interface defects, and most environment defects (46%) are caused by API defects. We found that training defects have diverse symptoms and root causes. We identified four main challenges in the DL reengineering process: model operationalization, performance debugging, portability of DL operations, and customized data pipeline. Integrating our quantitative and qualitative data, we propose a novel reengineering workflow.

Communicated by: Feldt and Zimmermann

Extended author information available on the last page of the article

Published online: 20 August 2024

Conclusions Our findings inform several conclusion, including: standardizing model reengineering practices, developing validation tools to support model reengineering, automated support beyond manual model reengineering, and measuring additional unknown aspects of model reengineering.

Keywords Empirical software engineering · Machine learning · Deep learning · Deep neural networks · Computer vision · Software reliability · Failure analysis · Bug study · Mixed methods · Case study

1 Introduction

Deep learning (DL) over neural networks achieves state-of-the-art performance on diverse tasks (Schmidhuber 2015), including games (Berner et al. 2019; Vinyals et al. 2019), language translation (Bahdanau et al. 2015; Wu et al. 2016), and computer vision (Ren et al. 2017; Redmon et al. 2016). After researchers demonstrate the potential of a DL approach in solving a problem, engineering organizations may incorporate it into their products.

This DL software reengineering task — of engineering activities for *reusing, replicating, adapting, or enhancing an existing DL model* using state-of-the-art DL approaches — is challenging for reasons including mismatch between the needs of research and practice (Tatman et al. 2018; Hutson 2018), variation in DL libraries (Pham et al. 2020) and other environmental aspects (Unceta et al. 2020), and the high cost of model training and evaluation (Goyal et al. 2018). An improved understanding of the DL engineering process will help engineering organizations benefit from the capabilities of deep neural networks.

As illustrated in Fig. 1, prior empirical studies have not fully explored the DL engineering process. These works have focused on understanding the characteristics of defects during DL development. These works consider defect characteristics and fix patterns, both in general (Humbatova et al. 2020; Sun et al. 2017; Zhang et al. 2020b) and by DL framework (Zhang et al. 2018; Islam et al. 2019). In addition, these works focused primarily on the "product" view of DL systems, which provides an incomplete view of software engineering practice.

Prior work conducted a literature review of existing failure studies and revealed a gap in the understanding of the causal chain of defects (Amusuo et al. 2022; Anandayuvaraj and Davis 2022). This gap suggests the need for failure analysis approaches that go beyond simply analyzing the product itself. A "beyond-the-product" interpretation of failures is needed to gain a more detailed understanding of how defects arise. To this end, a "process" view encapsulates the practices, activities, and procedures undertaken in the course of software development (Pressman 2005). It illuminates the steps followed, strategies employed, and challenges encountered during the process of creating software. While the "product" view focuses on the resulting defects in a DL system, the "process" view would allow us to understand the steps that led to those defects (Leveson 2016, 1995) and the practices that contributed to their resolution. This perspective also offers a deeper understanding of reengineering DL models, including how existing models are reused, adapted, or enhanced. It provides a framework to analyze not just the final product, but the entire journey of software development, offering more holistic insights.

Definition: We define **DL model reengineering process** as: engineering activities for *reusing, replicating, adapting, or enhancing an existing DL model*.

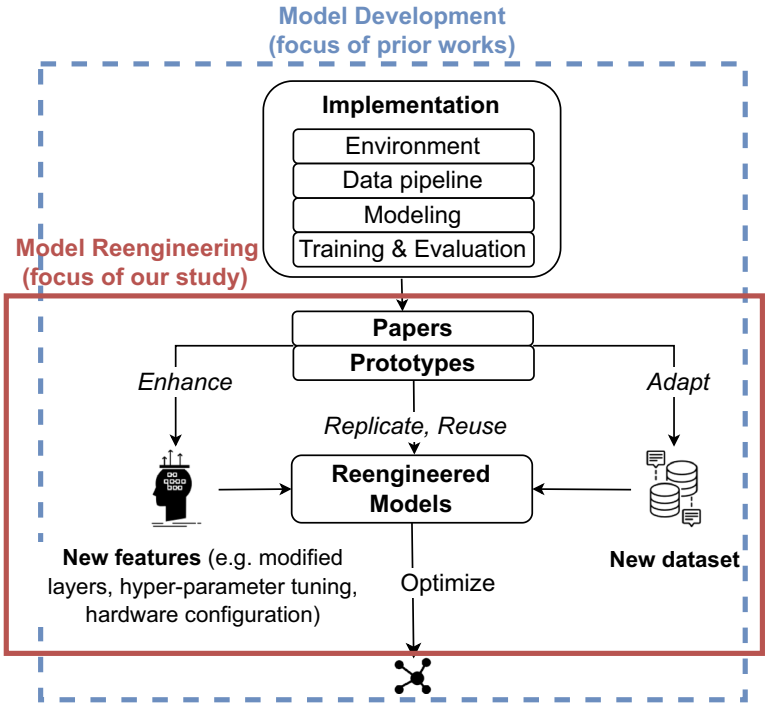


Fig. 1 High-level overview of a DL model development and application life cycle. Prior work (blue box) may have accidentally captured reengineering activities, but it did not describe reengineering activities as a distinct activity and was generally from "product" view. We specifically focus on model reengineering activities and "process" view (red box). Dashed lines implies that prior work does not distinguish these two kinds of activities. We define DL *reengineering* as reusing, replicating, adapting or enhancing existing DL models

In this paper, we examined the **Deep Learning reengineering process**: engineering activities for *reusing, replicating, adapting, or enhancing an existing DL model*. We scoped our work to Computer Vision, resulting in a case study of DL reengineering in a particular DL application domain (Ralph et al. 2021). We used a mixed-methods approach and drew from two complementary data sources (Johnson and Onwuegbuzie 2004). First, to explore the characteristics, challenges, and practices of CV reengineering, we analyzed 348 defects from 27 open-source DL projects (§5.1). Second, we describe the qualitative reengineering experiences of two open-source engineers and four industry practitioners, as well as a DL reengineering team (§5.2).

Combining these data, we report the challenges and practices of DL reengineering from a "process" view (§6). From our defect study, we observed that DL reengineering defects varied by DL stage (§6.1): environmental configuration is dominated by API defects (§6.4); the data pipeline and modeling stages are dominated by errors in assignment and initialization (§6.2); and errors in the training stage take diverse forms (§6.4). The performance defects discovered in the training stage appear to be the most difficult to repair (§6.3). From our interview study we identified similar challenges, notably in model implementation and in performance debugging (§6.5). These problems often arose from a lack of portability, *e.g.,* to different hardware, operating environment, or library versions. Interview subjects described their testing and debugging techniques to address these challenges. Synthesizing

these data sources, we propose an idealized DL reengineering workflow §7. The difficulties we identify in DL reengineering suggest that the community needs further empirical studies, and researchers should investigate more on DL software testing. Comparing to traditional reengineering and DL development, we suggest the necessity of developing techniques to support the reuse of pre-trained models and improve standardized practice.

In summary, our main contributions are:

- We conducted the first study that takes a "process" view of DL reengineering activities (§5).
- We quantitatively analyze 348 defects from 27 repositories in order to describe the characteristics of the defects in DL reengineering projects (§6.1–§6.4).
- We complement this quantitative failure study with qualitative data on reengineering practices and challenges (§6.5). Our subjects included 2 open-source contributors and 4 industry practitioners. To have a more comprehensive perspective, we also coordinated a two-year engineering effort by a student team to enrich this part of the study. To the best of our knowledge, this second approach is novel.
- We illustrate the challenges and practices of DL model reengineering with a novel reengineering process workflow. We propose future directions for theoretical, empirical, and practical research based on our results and analysis (§7).

2 Background and Related Work

2.1 Empirical Studies on Deep Learning Engineering Processes

DL models are being adopted across many companies. With the demand for engineers with DL skills far exceeding supply (Nahar et al. 2022), companies are looking for practices that can boost the productivity of their engineers. Google (Breck et al. 2017), Microsoft (Amershi et al. 2019), and SAP (Rahman et al. 2019) have provided insights on the current state of the DL development and indicate potential improvements. Breck *et al.* indicated that it is hard to create reliable and production-level DL systems (Breck et al. 2017). Amershi *et al.* proposed the requirements of model customization and reuse, *i.e.,* adapting the model on different datasets and domains (Amershi et al. 2019). Rahman *et al.* pointed out that knowledge transfer is one of the major collaboration challenges between industry and academia (Rahman et al. 2019). Our work identifies challenges and practices of knowledge transfer from academia to industry to help engineers create customized but reliable models.

In addition to views of the industry, academic researchers have conducted empirical studies and supplied strategies to solve some DL engineering challenges. Zhang *et al.* illustrated the need for cross-framework differential testing and the demand for facilitating debugging and profiling (Zhang et al. 2019). *Serban et al.* discussed the engineering challenges and the support of development practices, and highlighted some effective practices on testing, automating hyper-parameter optimization, and model selection (Serban et al. 2020). Lorenzoni *et al.* showed how DL developers could benefit from a traditional software engineering approach and proposed improvements in the ML development workflow (Lorenzoni et al. 2021). These studies and practices were based on "products" (*e.g.,* engineers who develop DL components) rather than "process" (a specific class of engineering work). Our focus is on a particular engineering process (DL reengineering) rather than considering all DL engineering work. This perspective allowed us to bring novel insights specific to DL reengineering, including common failure modes and problem-solving strategies.

Figure 1 illustrates how our work differs from previous empirical studies. Although there have been many software engineering studies on DL, they collect data from a "product" view of DL systems (Islam et al. 2019; Shen et al. 2021; Chen et al. 2022b; Lorenzoni et al. 2021). These works sample open-source defects or Stack Overflow questions and report on the features, functionalities, and quality of DL software. Some work mentioned specific reengineering activities, such as model customization (Amershi et al. 2019) and knowledge transfer of DL technologies (Rahman et al. 2019). However, there is no work focusing on the "process" of these reengineering-related activities. We conduct the first empirical study of DL software from a "process" view which focuses on the activities and processes involved in creating the software product. We specifically examine the DL reengineering activities and process by sampling the defects from open-source research prototypes and replications. We also collected qualitative data about the activities and process by interviewing open-source contributors and leaders of the student reengineering team.

2.2 Reengineering in Deep Learning

Historically, *software reengineering* referred to the process of replicating or improving an existing implementation (Linda et al. 1996). Engineers undertake the reengineering process for needs including optimization, adaptation, and enhancement (Jarzabek 1993; Byrne 1992; Tucker and Devon 2010). Today, we specifically observe that DL engineers are reusing, replicating, adapting, and enhancing existing DL models, to understand the algorithms and improve their implementations (Amershi et al. 2019; Alahmari et al. 2020. The maintainers of major DL frameworks, including TensorFlow and PyTorch, store reengineered models within official GitHub repositories (Google 2022; Meta 2022). Many engineering companies, including Amazon (Schelter et al. 2018), Google (Kim and Li 2020), and Meta (Pineau 2022) are likewise engaged in forms of *DL reengineering*. For example, Google sponsors the TensorFlow Model Garden which provides "a centralized place to find code examples for state-of-the-art models and reusable modeling libraries" (Kim and Li 2020). Many research papers use PyTorch because it is easy to learn and has been rapidly adopted in research community, but many companies have preferred TensorFlow versions because of available tools (*e.g.,* for visualization) and robust deployment (O'Connor 2023). Our work describes some of the differences between traditional software reengineering and DL reengineering process, in terms of the goals and its causal factors.

In this work we conceptualize reengineering a DL model as a distinct engineering activity from developing a new DL model. Reengineering involves refining an existing DL model implementation for improved performance, such as modifying architecture or tuning hyper-parameters. Conversely, developing a new DL model usually involves building a model from scratch, potentially drawing upon high-level concepts or algorithms from existing models but not directly modifying their code. While the distinction is useful, the boundary can blur. A prime example of the blurred boundary is seen in ultralytics/yolov5, where significant enhancements to a replication of YOLOv3 codebase led to models considered "new" (the YOLOv5 model) (Nepal and Eslamiat 2022). In the context of this study, our definition of reengineering includes adaptation and enhancement, which can sometimes be considered as developing a new DL model, specifically when a model from one domain is adapted to another domain. An example is the adaptation of an R-CNN model for small object detection (Chen et al. 2017). In our study, such adaptation is considered as part of the reengineering process. Although there is a overlap between reengineering and developing new models, our study

reveals unique challenges and practices specific to reengineering activities. We discuss these distinct findings in our study on reengineering process in §7.2.1.

Reengineering in DL software has received much attention in the research community (Bhatia et al. 2023; Hutson 2018; Pineau et al. 2020; Gundersen and Kjensmo 2018; Gundersen et al. 2018). Pineau *et al.* noted three needed characteristics for ML software: reproducibility, re-usability, and robustness. They also proposed two problems regarding reproducibility: an insufficient exploration of experimental variables, and the need for proper documentation (Pineau et al. 2020). Gundersen *et al.* surveyed 400 AI publications, and indicated that documentation facilitates reproducibility. They proposed a reproducibility checklist (Gundersen and Kjensmo 2018; Gundersen et al. 2018). Chen *et al.* highlights that the reproducibility of DL models is still a challenging task as of 2022 (Chen et al. 2022a). Consequently, the characteristics of the reengineering process are significant for both practitioners (Villa and Zimmerman 2018; Pineau 2022) and researchers (MLR 2020; Ding et al. 2021). Previous research on machine learning research repositories on GitHub explored contributions from forks by analyzing the commits and PRs, but found few actually contributed back (Bhatia et al. 2023). Our study takes a different data collection approach by examining defect reports from downstream users in the upstream repository, offering a new perspective on the reengineering process and feedback from downstream engineers. The DL reengineering process and its associated challenges and practices have been unexplored. Prior work has used the concept of DL reengineering as a particular method of reusing DL models (Qi et al. 2023). Our research uniquely defines and investigates the process of DL model reengineering. This reengineering process and its associated challenges and practices have been previously unexplored in prior work, but we highlight its significance in the software engineering field.

The combination of the software, hardware, and neural network problem domains exacerbates the difficulty of deep learning reengineering. The DL ecosystem is evolving, and practitioners have varying software environments and hardware configurations (Boehm and Beck 2010). This variation makes it hard to reproduce and adapt models (Goel et al. 2020). Additionally, neural networks are reportedly harder to debug than traditional software, *e.g.,* due to their lack of interpretability (Bibal and Frénay 2016; Doshi-Velez and Kim 2017). To facilitate the reengineering of DL systems, researchers advise the community to increase the level of portability and standardization in engineering processes and documentation (Gundersen and Kjensmo 2018; Pineau et al. 2020; Liu et al. 2020). Microsoft indicated that existing DL frameworks focus on runtime performance and expressiveness and neglect composability and portability (Liu et al. 2020). The lack of standardization makes finding, testing, customizing, and evaluating models a tedious task (Gundersen et al. 2018). These tasks require engineers to "glue" libraries, reformat datasets, and debug unfamiliar code — a brittle, time-consuming, and error-prone approach (Sculley et al. 2014). To support DL (re-)engineers, we conducted a case study on the defects, challenges, and practices of DL reengineering.

2.3 The Reuse Paradigm of Deep Learning Models

DL model reuse has become an important part of software reuse in modern software systems. Concurrent empirical work reveals that engineers frequently employ pre-trained DL models to minimize computational and engineering costs (Jiang et al. 2023b) In their review, (Davis et al. 2023) describe three categories of DL model reuse: conceptual, adaptation, and deployment (Davis et al. 2023). Each category presents unique complexities and requirements, which can complicate the reuse process and lead to various challenges. In those

terms, DL model reengineering occurs during conceptual and adaptation reuse. Panchal *et al.* have documented the difficulties in reengineering and deploying DL models in real-world products, which often result in wasted time and effort (Panchal et al. 2023, 2024b). Furthermore, adapting DL models for constrained environments, such as in DL-based IoT devices, is particularly challenging due to reduced performance, memory limitations, and a lack of reengineering expertise (Gopalakrishna et al. 2022). Currently, there is no systematic study on understanding the challenges during the DL model reuse and reengineering process, nor a comprehensive exploration of how these models can be effectively adapted and optimized for varying deployment contexts.

To support the model reuse process more effectively and reduce engineering costs, the community has developed model zoos, or hubs, which are collections of open-source DL models. These resources are categorized into platforms and libraries (Jiang et al. 2022a). Model zoo platforms like Hugging Face (Wolf et al. 2020), PyTorch Hub (Pytorch 2021), TensorFlow Hub (Ten 2021), and Model Zoo (Jing 2021) provide engineers with an organized and effective way to locate suitable open-source model repositories via a hierarchical search system (Jiang et al. 2023b, b). Additionally, model zoo libraries developed by companies, such as *detectron* from (Meta 2024a, b), *mmdetection* from OpenMMLab (Chen et al. 2019), and *timm* from Hugging (Face 2024), offer more direct access to reusable resources. However, the specific challenges associated with reusing individual open-source models or those aggregated in model zoos remain underexplored.

Prior research underscores that, at a system level, the reuse process of DL models diverges significantly from traditional software engineering practices, where reuse often entails direct, copy-paste methods across projects (Jiang et al. 2023b). Such traditional approaches allow developers to replicate entire software architectures and functionalities from one project to another with minimal modifications (Gharehyazie et al. 2017). However, due to their complex dependencies and specialized configurations, DL models require significant adaptation to meet the demands of new applications and environments (Gopalakrishna et al. 2022; Jajal et al. 2023; Jiang et al. 2023b). For example, downstream tasks frequently need different datasets or hardware, necessitating a comprehensive reengineering process for effective DL model reuse. To illuminate engineers' considerations and challenges in DL model reuse process, in this paper we conduct the first detailed study of DL reengineering activities.

Code forking remains one of the major reuse methods in the DL reuse paradigm. Bhatia *et al.* empirically studied 1,346 ML research repositories and their 67,369 forks (the median repository had 8 forks), indicating that forking of ML and DL projects is common (Bhatia et al. 2023). Engineers often adapt model training and serving pipelines, as well as reusable model infrastructures in production, further indicating that code reuse remains a primary method for DL model reuse (Panchal et al. 2024b). These practices persist despite the existence of multiple model reuse approaches such as fine-tuning and transfer learning (Käding et al. 2017; Tan et al. 2018), indicating contexts where code-level reuse is preferred to model-level reuse. Our work studies DL reengineering from a process perspective, based on forking as a major reuse method.

2.4 Deep Learning Defects and Fix Patterns

Prior work focused on the general DL development process, studying the defects, characteristics, and symptoms. Islam *et al.* demonstrated DL defect characteristics from 5 DL libraries in GitHub and 2716 Stack Overflow posts (Islam et al. 2019). Humbatova *et al.* analyzed data from Stack Overflow and GitHub to obtain a taxonomy of DL faults (Humbatova et al.

2020). By surveying engineers and researchers, Nikanjam *et al.* specified eight design smells in DL programs (Nikanjam and Khomh 2021).

Furthermore, researchers conducted works on DL fix patterns (Sun et al. 2017; Islam et al. 2020). Sun *et al.* analyzed 329 defects for their fix category, pattern, and duration (Sun et al. 2017). Islam *et al.* considered the distribution of DL fix patterns (Islam et al. 2020). Their findings revealed a distinction between DL defects and traditional ones. They also identified challenges in the development process: fault localization, reuse of trained models, and coping with frequent changes in DL frameworks. Some development defects studied from prior work are "wrong tensor shape" (Humbatova et al. 2020) and "`ValueError` when performing `matmul` with TensorFlow" (Islam et al. 2019). Our study offers a "process" perspective on the reengineering process. This viewpoint has enabled us to identify distinct types of defects, as derived from issues such as "user's training accuracy was lower than what was claimed" or "training on a different hardware is very slow" (cf. Table 3).

Prior work considered the defects in the DL model development process, without distinguishing which part of the development process they were conducting. In this work, we focus on defects arising specifically during the DL model reengineering process (Fig. 1). We use defect data from GitHub repositories. We also collect interview data, providing an unusual perspective — prior studies used data from Stack Overflow (Islam et al. 2019; Zhang et al. 2018; Humbatova et al. 2020), open-source projects (Zhang et al. 2018; Islam et al. 2019; Sun et al. 2017; Humbatova et al. 2020; Shen et al. 2021; Garcia et al. 2020), and surveys (Nikanjam and Khomh 2021).

3 Research Questions and Justification

3.1 Research Questions in light of the Previous Literature

To summarize the literature: Prior work has studied the problems of DL engineering, *e.g.,* considering the characteristics of DL defects (Zhang et al. 2020c) sometimes distinguished by DL framework (Zhang et al. 2018). DL reengineering is a common process in engineering practice, but is challenging for reasons including (under-)documentation, shifting software and hardware requirements, and unpredictable computational expense. Prior work has not distinguished between the activities of DL development and DL reengineering, and has not examined DL reengineering specifically.

Goal: The goal of this work is to provide the first systematic study of the characteristics, challenges, and practices of DL model reengineering.
In this work, we define the **DL model reengineering process** as: engineering activities for *reusing, replicating, adapting, or enhancing an existing DL model*.

We adopt a mixed-method approach to explore the challenges and characteristics of DL reengineering. We integrate quantitative and qualitative methodologies to provide a comprehensive understanding of the topic. We ask:

Theme 1: Quantitative – Understanding defect characteristics

RQ1 *What defect manifestations are most common in Deep Learning reengineering, in terms of project type, reporter type, and DL stages?*
RQ2 *What types of general programming defects are most frequent during Deep Learning reengineering?*

RQ3 *What are the most common symptoms of Deep Learning reengineering defects and how often do they occur?*
RQ4 *What are the most common Deep Learning specific reengineering defect types?*

Theme 2: Qualitative – Underlying challenges and practices

RQ5 *When software engineers perform Deep Learning reengineering, what are their practices and challenges?*

3.2 How the Answers are Expected to Advance the State of Knowledge

Here are the contributions made by each theme.

- **Theme 1**: The *quantitative* theme, addressed through RQ1-4, focuses on the typical characteristics of DL reengineering defects, aiming to establish a baseline of empirical data on how defects manifest across various project types, reporter types, and DL stages. Like previous failure studies on DL software (Islam et al. 2019; Zhang et al. 2018; Shen et al. 2021; Chen et al. 2022b; Jajal et al. 2023), this part of the study maps the landscape of DL reengineering defects to foundational knowledge that guides subsequent defect management and mitigation strategies. The detailed insights gained from analyzing the types and symptoms of defects (RQ2-4) pinpoint specific areas where developers and researchers can focus their efforts.
- **Theme 2**: The *qualitative* theme, in RQ5, seeks the "secret life of bugs" (Aranda and Venolia 2009; Baysal et al. 2012). We leverage insights from the quantitative theme to design a qualitative study — interviews designed to examine challenges not captured in artifacts such as open-source software repositories. By doing so, we bridge the gap between empirical data and the underlying human experiences, offering deeper, context-rich insights that provide a nuanced understanding of the challenges faced in DL reengineering.

Together, these integrated quantitative and qualitative themes provide an empirical understanding of current DL reengineering practices and challenges.

To scope this research, we next present a model of DL reengineering as practiced in open-source software (§4). Then, we outline our mixed-method approach to understanding DL reengineering (§5), with quantitative §5.1 and qualitative §5.2.1 methodologies). Results for each theme are in §6, with a synthesis in §7.

4 Model of Deep Learning Reengineering

As discussed in §2.2, the expected process of software reengineering includes replicating, understanding, or improving an existing implementation (Linda et al. 1996). This process is usually used for optimization, adaptation, and enhancement (Jarzabek 1993; Byrne 1992; Tucker and Devon 2010). Software reengineering has not previously been used as a lens for understanding DL engineering behavior. In this section we introduce a model of DL reengineering, including the concepts of DL reengineering repositories (§4.1), and the concepts of reengineering defect characteristics (§4.2).

4.1 Concepts of DL Reengineering Repositories

Here we present the main concepts used in this study: relationships of open-source reengineering projects §4.1.1, and reengineering repository types (§4.1.2).

4.1.1 Relationships Between DL Reengineering Repositories

Based on the typical open-source software collaboration process (Fitzgerald 2006), we modeled the expected relationship between DL reengineering repositories — see Fig. 2. This figure depicts the relationship between three types of projects:

- *Research prototype*: an original implementation of a model.
- *Replications*: a replication and possible evolution of the research prototype.
- *User "fork"*: a GitHub fork, clone, copy, etc. where users try to reuse or extend a model of the previous two types.

We expect the existence of two actions, "*REUSE* (fork)" and "*REPORT (defect)*", happening between different project types. The down-stream projects reuse up-stream projects, and engineers report new issues when they encounter defects or identify necessary enhancements. Typically the issues are reported in upstream projects, the maintainers explain the necessary fix patterns for reengineers, and the defects are repaired in downstream ones. Our work concentrated on popular upstream projects, such as research prototypes and replications, because these repositories are widely used and referenced and they contain a large number of issues. Recognizing the limitations of using upstream repositories as a data source, we supplemented our failure analysis with an interview study involving downstream users to provide a more comprehensive perspective (§5.2).

4.1.2 DL Reengineering Repository Types

Based on our analysis of GitHub repositories (§5.1.2), we distinguish two types of DL repositories: *zoo* repositories and *solo* repositories. *Zoo repositories* all contain implementations of several models (Tsay et al. 2020). A *zoo repository* can be the combination of research

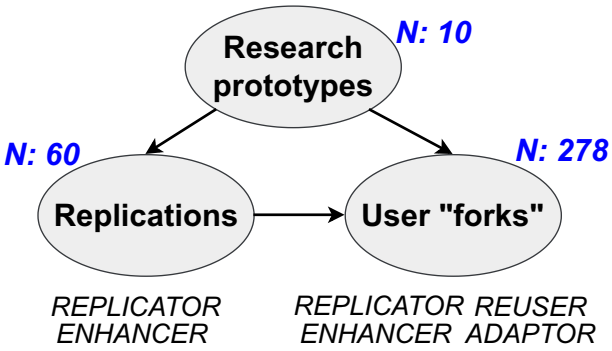


Fig. 2 Relationship between CV reengineering repositories. The capitalized annotations indicate what types of reporters commonly open issues in the above projects (cf.Table 1). Arrows indicate the dataflow of model reuse between upstream and downstream projects. N: the number of defects resolved in each type of the projects we analyzed

Table 1 Reporter types in the DL reengineering ecosystem. The reporter types are determined by whether they use the same code, dataset, and algorithm compared to the upstream project. The categories and descriptions in this table are derived from our literature review. The source of each category is indicated as a reference

Reporter Category	Description
Re-user (Amershi et al. 2019; Alahmari et al. 2020)	Uses the same code and dataset. This is the common-case behavior associated with pure re-use.
Adaptor (Li et al. 2020)	Adapts code to other tasks (different dataset) and finds inconsistency compared to expectations.
Enhancer (Git 2020)	Adds new features (*e.g.,* layer modification, hyperparameter tuning, multi-GPU training configuration).
Replicator (Alahmari et al. 2020; Pineau et al. 2020)	Uses the same algorithm, data, and configuration, in distinct implementation (*e.g.,* TensorFlow vs. PyTorch).

prototypes and replications. For example, `TensorFlow Model Garden` contains different versions of YOLO (Google 2022). These *zoo repositories* have been widely reused and contain many relevant issues (Table 4). In this work, we define a *solo* repository as either the research prototype from the authors of a new model or an independent replication.

4.2 Concepts of DL Reengineering Defects

Prior work shows that defect type(s), symptom(s), and root cause(s) are useful dimensions for understanding a defect (Islam et al. 2019; Wang et al. 2017; Liu et al. 2014). To better characterize DL reengineering defects, we first introduce the relevant concepts specific to the DL reengineering process — defect *reporters* (§4.2.1, Table 1) and *reengineering phases* (§4.2.2, Table 2).

4.2.1 DL Reengineering Defect Reporters

Table 1 defines four different types of *defect reporters* based on their reengineering activities. Prior work indicates some inconsistency in the meaning of these terms (Gundersen and Kjensmo 2018), so we state here the definitions used in our study. Our definitions are based

Table 2 Reengineering phases and relevant definitions, determined by the runnability of code and the data used to train the model. The categories and descriptions in this table are derived from our literature review. The source of each category is indicated as a reference

Defect Category	Description
Basic defects (Islam et al. 2019; Zhang et al. 2018; Guan et al. 2023)	The code does not run (*e.g.,* it crashes, behaves very incorrectly, or runs out of memory).
Reproducibility defects (Sculley et al. 2015; Liu et al. 2021)	Using the same data, the code runs without basic defects, but does not match the documented performance (*e.g.,* accuracy, latency).
Evolutionary defects (Git 2020)	The code and/or data has been changed to adapt to the user's needs. It runs without basic defects, but does not match the specification/desired performance.

on prior work (Pineau et al. 2020; Li et al. 2020; Git 2020; Amershi et al. 2019; Alahmari et al. 2020).

4.2.2 DL Reengineering Phases

Table 2 defines three types of *reengineering phases*. We followed prior work studying the phases of defects in terms of activities (Saha et al. 2014). The most obvious phase during reengineering process is the model crashing when it is executed. This is a common defect when running any DL model (Islam et al. 2019; Zhang et al. 2018; Guan et al. 2023). Prior work has also noted the difficulty in reproducing DL models, and we would like to understand how these difficulties impact the reengineering process (Sculley et al. 2015; Liu et al. 2021). Additionally, there are also a significant number of issues related to performance improvement or new feature addition to the original implementation. Often labelled as "enhancement" on GitHub, these issues typically emerge when features are added or when environments change[1] (Git 2020). Since enhancements form an integral part of the reengineering process, we included this as a category of the reengineering phase. The reengineering phases provide a high-level view of the reengineering process by combining defect types, root causes, and relevant contexts.

5 Methodology

To answer our research questions, we used a mixed-method approach (Johnson and Onwuegbuzie 2004) which provides both quantitative and qualitative perspectives. For RQ1–RQ4, we conducted a quantitative failure analysis. Systematically characterizing and critiquing engineering defects is a crucial research activity (Amusuo et al. 2022). We specifically analyzed DL reengineering defects in open-source reengineering projects. However, we acknowledge that this data source may be constraining because GitHub issues are known to be limited in the breadth of issue types and the depth of detail (Aranda and Venolia 2009). The role of RQ5 is to address this limitation through a qualitative complement. To answer RQ5, we designed an interview study, which was informed by the findings from the failure analysis, and collected qualitative reengineering experiences from open-source contributors and from the leaders of a student DL reengineering team. These two perspectives are complementary: the open-source defects gave us broad insights into some kinds of reengineering defects, and the interview data gives us deep insights into the reengineering challenges and process. Figure 3 shows the relationship between our questions and methods. To promote replicability and further analysis, all data is available in our artifact (§10).

Deep learning is a broad field with many sub-topics, including representation learning and generative models (Goodfellow et al. 2016). At present the primary application domains are (1) computer vision (*e.g.,* for manufacturing (Wang et al. 2018) and for autonomous vehicles (Kuutti et al. 2020)); and (2) natural language processing (*e.g.,* automatic translation (Popel et al. 2020) and chatbots such as ChatGPT (Taecharungroj 2023)). One study cannot hope to examine reengineering in all of these sub-areas. To scope our study, we applied a *case study* approach (Perry et al. 2004; Runeson and Höst 2009). This case study approach gives us a comprehensive view of a particular DL application domain. Our phenomenon of

[1] Here are two examples of enhancement for environment changes: (1) *torch/vision #2148* improved error message when newer versions of Torchvision or PyTorch were used; (2) *tensorflow/models #1251* fixed issues caused by a newer Python version.

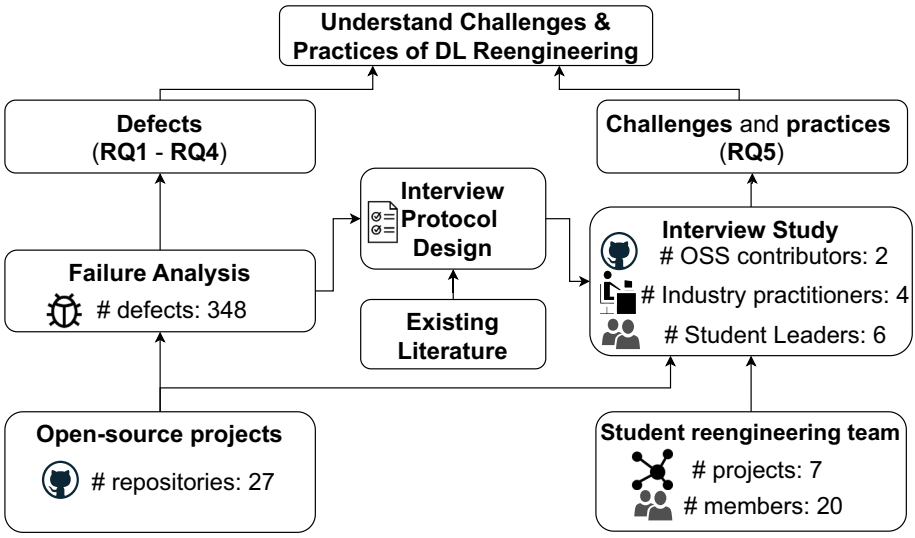


Fig. 3 Relationship of research questions to methodology. The failure analysis is conducted on open-source GitHub projects that undertake DL reengineering. The interview study is conducted on 2 contributors to the GitHub projects we studied, 4 industry practitioners who work on DL reengineering (recruited via social media platforms), and 6 leaders from our student DL reengineering team

interest was DL reengineering, and we situated the study in the context of DL reengineering for *Computer Vision (CV)*. CV plays a critical role in our society, with applications addressing a wide range of problems from medical to industrial robotics (Xu et al. 2021). Although case studies have limited generalizability, we suggest three reasons why our results may generalize to other DL application domains.

1. As a major application domain of DL techniques (Voulodimos et al. 2018; CVE 2021; Xu et al. 2021), CV serves as a microcosm of DL engineering processes (Amershi et al. 2019; Thiruvathukal et al. 2022).[2]
2. Techniques are shared between computer vision and other applications of deep learning. For example, the computer vision technique of Convolution Neural Networks (CNNs) (O'Shea and Nash 2015) has been adapted and applied to identify features and patterns in Natural Language Processing and Audio recognition tasks (Li et al. 2021; Alzubaidi et al. 2021). From other fields, neural network architectures such as transformers and large language models were initially designed for Natural Language Processing (NLP) and are now being applied to CV tasks (Cheng et al. 2022; Zou et al. 2023).
3. The engineering process for deep learning models, encompassing stages such as problem understanding, data preparation, model development, evaluation, and deployment, is consistent across different domains, including both computer vision and natural language processing, thus further supporting the potential generalizability of our findings (Sculley et al. 2015; Amershi et al. 2019)

Additionally, our focus aligns with prior empirical software engineering studies of deep learning, much of which has concentrated on the field of computer vision as a representative

[2] We acknowledge that there are approaches to computer vision that do not leverage deep learning (Szeliski 2022; Forsyth and Ponce 2002). In this study, we focused on engineers applying deep learning to problems in computer vision.

domain for deep learning studies (Ma et al. 2018a, b; Wei et al. 2022; Pham et al. 2020). Our case study will thus provide findings for computer vision reengineering projects, which are important; and our results may generalize to other domains of deep learning. However, all case studies provide depth in exchange for breadth, and so some aspects of DL reengineering in the context of computer vision may not generalize (Perry et al. 2004; Runeson and Höst 2009). For example, datasets, architectures, and problem definitions are different (Tajbakhsh et al. 2020; Shrestha and Mahmood 2019; Khamparia and Singh 2019) and so the necessary steps for data pipeline modification, architectural adaptation, and replication checks may differ (§8).

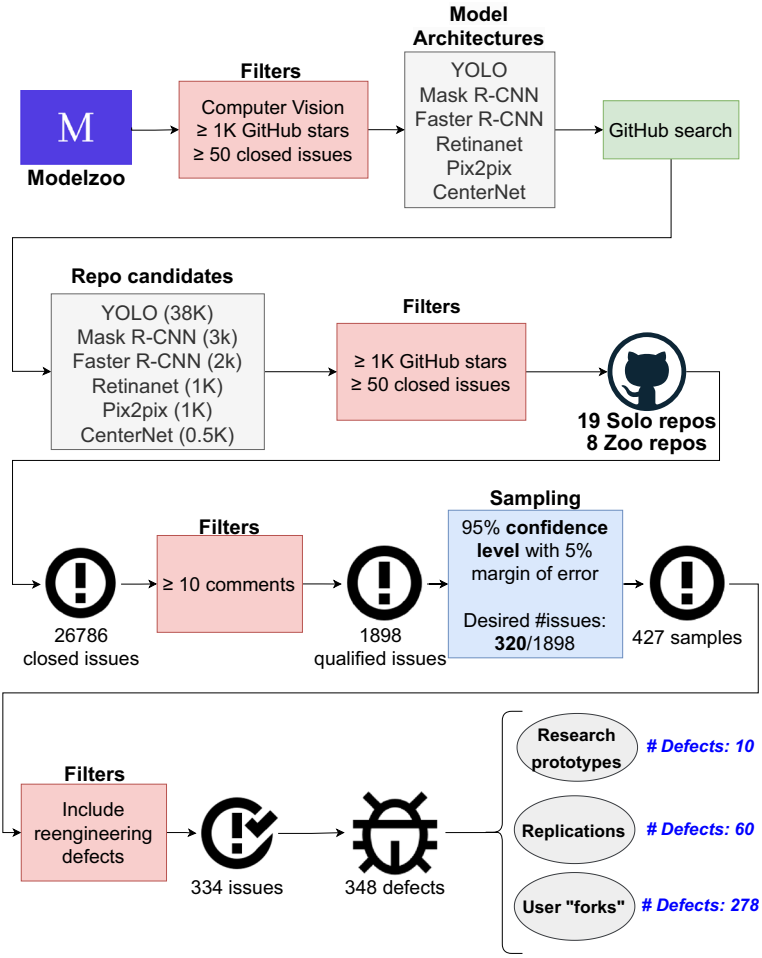


Fig. 4 Overview of defect collection and distribution of the collected defects in three project types, as well as the number of projects and defects we got in each step. The number of each model architecture after GitHub search (Most data were collected in 2021. The number of stars associated with "Repo candidates" were collected in Feb. 2023.)

5.1 RQ1-4: Failure Analysis on Computer Vision Reengineering

In designing our failure analysis ("bug study"), we adopted the guidance from Amusuo *et al.*, which provides a structured approach to examining failures. This framework includes several key stages: defining the problem scope (§5.1.1), collecting defects (§5.1.2, §5.1.3), and analyzing these defects (§5.1.4) (Amusuo et al. 2022). We also adapted previous failure analysis methods (Islam et al. 2019; Zhang et al. 2018; Wang et al. 2020a; Eghbali and Pradel 2020) to identify and analyze defects in DL reengineering. Figure 4 shows the overview of our data collection in the failure analysis.

5.1.1 Problem Scope

In our work, we focused on defect (Table 3) manifestation, general programming types, defect symptoms, and DL-specific defect types. In total, we examined 348 defects from 27 repositories (Table 4). The rest of this section discusses our selection and analysis method.

5.1.2 Repository Selection

To find sufficient DL reengineering defects in this target — research prototypes and replication projects — we chose to look at CV models with substantial popularity and GitHub issues. We proceeded as follows, along the top row of Fig. 4:

1. We started by selecting popular CV models from ModelZoo (Jing 2021), a platform that tracks state-of-the-art DL models. We looked at relevant GitHub repository and chose the CV models with over 1K GitHub stars and 50 closed issues.
2. We searched for implementations of these model architectures on GitHub. We utilized GitHub's code search feature, filtering by keywords associated with the model names.
3. For each architecture, we selected projects that implement the same model, each with a minimum of 1,000 GitHub stars and 50 closed issues. The projects were selected based on their popularity, as indicated by the number of stars (Borges and Valente 2018). However, to maintain the diversity of our data, we only chose up to the top five repositories for any given model. If there were more than five projects for a single model, we limited our selection to the five most popular ones. This approach helps to prevent bias in our results, such as over-representation of specific reproducibility issues or enhancements tied to a single model or architecture family. If there was only one implementation matching our criteria, we excluded that model in our analysis.

As shown in Table 4 (last 4 rows), for two architectures this process did not always yield both prototypes and replications. For `Pix2pix` we retrieved two prototypical implementations in different programming languages (*viz.* Lua and Python). For `CenterNet` we retrieved two prototypes for different model architectures that share the same model name. However, on inspection we found that these four repositories were actively reused, had many people engaged in the reengineering process, and had many reengineering defects. Therefore, we included the issues from these repositories in our analysis.

The same model architecture can be found in either *zoo repositories* or *solo repositories* during GitHub searching (§4.1.2). Most of the repositories (19/27) we identified during the GitHub searching are *solo repositories* which only implement a single model. Both *solo* and *zoo repositories* have DL reengineering defects reported by down-stream replicators or users. Therefore, we applied the same data collection methods to them and put the data together.

Overall, we examined 19 *Solo Repositories* and 8 *Zoo Repositories*.

Table 3 Examples of reengineering defects included in our study, plus non-reengineering defects and non-defects that were both excluded from the study. A reengineering defect refers to an error or flaw that occurs during the process of reusing, replicating, adapting, or enhancing an existing deep learning model

Issue ID	Type	Description	Fix
tensorflow/models #6043	Reeng.	The losses occasionally contained a NaN value when using a customized dataset.	Replace the empty sequence in tensor with zeros.
ultralytics/yolov3 #310	Reeng.	User training accuracy was lower than what was claimed by the replicator in the documentation.	Load the original checkpoint weights before training.
matterport/Mask_RCNN #1938	Reeng.	Training on the RTX 2080 Ti GPU with CUDA library version 9 is slow.	Fix the environment by following provided instructions and reduce the training epochs.
facebookresearch/ detectron2#3225	Reeng.	Documentation does not seem to have been updated to reflect the new config files (`.py` rather than `.yaml`)	Add documentation for new training script to use existing configuration files.
facebookresearch/ Detectron #370	Non-reeng. defect	The `Caffe2` library does not have GPU support.	Use Docker installation.
tensorflow/models #1838	Non-reeng. defect	`pip install` does not work to install a specific version of TensorFlow.	Install the package from PyPI.
xingyizhou/ CenterNet #241	Non-defect	The user asked about the implementation details of a loss function.	N/A
facebookresearch/ detectron2 #8	Non-defect	The user requested ONNX support of an existing model.	N/A

Table 4 The studied repositories and defects. The *Closed issues (qualified)* indicates the total closed issues and those matching two filters (closed, ≥ 10 comments). *Samples* are sampled from the qualified closed issues. After these selection criteria, we manually identified 334 GitHub issues from the samples that described at least one reengineering defect. Some GitHub issues contained multiple reengineering defects, causing some of the counts to increase between the number of reengineering issues and defects. †Repository `rwightman/pytorch-image-models` was converted to `huggingface/pytorch-image-models` after the study completed

Repository	Type	Stars (K)	Forks (K)	Closed issues (qualified)	Samples	Reeng. issues (defects)
Zoo Repos						
tensorflow/models *(Google)*	Zoo	71.8	45.0	5560 (580)	58	51 (51)
facebookresearch/Detectron *(Facebook)*	Zoo	24.8	5.4	613 (27)	20	15 (15)
facebookresearch/detectron2 *(Facebook)*	Zoo	18.7	5.0	2605 (90)	20	12 (13)
open-mmlab/mmdetection	Zoo	17.1	6.1	4348 (155)	20	16 (16)
rwightman/pytorch-image-models†	Zoo	14.3	2.3	414 (11)	11	5 (5)
pytorch/vision *(Facebook)*	Zoo	10.2	5.3	1504 (139)	20	14 (14)
NVIDIA/DeepLearningExamples *(NVIDIA)*	Zoo	6.7	2.0	436 (22)	20	18 (18)
qubvel/segmentation_models	Zoo	3.5	0.8	167 (12)	12	8 (8)
YOLO						
ultralytics/yolov5	Solo (Proto.)	18.2	6.3	3795 (279)	20	19 (21)
ultralytics/yolov3	Solo (Repl.)	8.0	3.0	1671 (204)	20	17 (21)
qqwweee/keras-yolo3	Solo (Repl.)	6.9	3.4	226 (16)	16	13 (13)
eriklindernoren/PyTorch-YOLOv3	Solo (Repl.)	6.3	2.5	557 (26)	20	18 (19)
YunYang1994/tensorflow-yolov3	Solo (Repl.)	3.5	1.4	116 (3)	3	3 (4)
Mask R-CNN						
matterport/Mask_RCNN	Solo (Proto.)	20.9	10.2	783 (62)	20	15 (16)
CharlesShang/FastMaskRCNN	Solo (Repl.)	3.1	1.1	55 (2)	2	1 (1)
TuSimple/mx-maskrcnn	Solo (Repl.)	1.8	0.5	83 (7)	7	6 (6)

Table 4 continued

Repository	Type	Stars (K)	Forks (K)	Closed issues (qualified)	Samples	Reeng. issues (defects)
Faster R-CNN						
ShaoqingRen/faster_rcnn	Solo (Proto.)	2.5	1.2	53 (5)	5	4 (4)
rbgirshick/py-faster-rcnn	Solo (Repl.)	7.5	4.1	253 (33)	20	17 (17)
jwyang/faster-rcnn.pytorch	Solo (Repl.)	6.6	2.2	363 (34)	20	17 (18)
endernewton/tf-faster-rcnn	Solo (Repl.)	3.6	1.6	65 (15)	15	12 (12)
chenyuntc/simple-faster-rcnn-pytorch	Solo (Repl.)	3.4	1.0	267 (2)	2	1 (2)
Retinanet						
fizyr/keras-retinanet	Solo (Proto.)	4.2	2.0	1227 (90)	20	16 (18)
yhenon/pytorch-retinanet	Solo (Repl.)	1.8	0.3	78 (4)	4	2 (2)
pix2pix						
junyanz/pytorch-CycleGAN-and-pix2pix	Solo (Proto.)	16.2	4.9	847 (48)	20	15 (15)
phillipi/pix2pix	Solo (Proto.)	8.7	1.6	119 (13)	13	5 (5)
CenterNet						
xingyizhou/CenterNet	Solo (Proto.)	6.0	1.7	542 (17)	17	12 (12)
Duankaiwen/CenterNet	Solo (Proto.)	1.8	0.6	68 (2)	2	2 (2)
Total				**26786 (1898)**	**427**	**334 (348)**

5.1.3 Issue Selection

As shown in the middle row of Fig. 4, we applied two filters on issues in these repositories: (1) Issue status is *closed* with an associated *fix*, to understand the defect type and root causes (Tan et al. 2014); (2) Issue has $\geq$ 10 comments, providing enough information for analysis. We first used the two filters to filter the full set of issues in each repository, and then sampled the remainder. After applying these filters, there were 1898 qualifying issues across 27 repositories.

We have two goals during our issue selection. **First**, on the resulting distributions, we want to achieve a confidence level of at least 95% with 5% margin of error. To reach that goal, our sample must include at least 320 issues from the 1898 qualifying issues.[3] **Second**, we want to analyze at least 10% of the issues for each reengineering project, but this was balanced against the wide range of qualifying issues for each repository. For example, `ultralytics/yolov5` has 279 qualifying issues while `yhenon/pytorch-retinanet` has only 4. We first conducted a pilot study on 5 solo projects which includes 79 defects.[4] The pilot study indicated that choosing the 20 most-commented issues would cover roughly 10% of the issues for these projects and give us plenty of data for analysis. Our sampling approach resulted in 427 qualified issues that match these sampling criteria, and we identified 334 reengineering issues (cf. Table 4). The number of samples obtained is above the required size for the desired confidence level. We observed that some GitHub issues contained multiple reengineering root causes and fixes. We treated them as distinct defects. Finally, we identified 348 reengineering defects.

In order to achieve a sufficient confidence level from analyzing only 10% of issues, we applied our filters uniformly across both solo and zoo repositories to prevent bias stemming from specific models or zoos. This strategy increases our confidence that the insights derived from this sample are reflective of the population of DL reengineering defects.

For two zoo repositories (`tensorflow/models` and `pytorch/vision`), issues having the most comments are primarily controversial API change requests, not reengineering defects, so we randomly sampled 10% of the closed issues instead. For some smaller repositories, taking the top-20 qualifying issues consumed all available defects (Table 4).

For most of the selected repositories, we sorted the remaining issues by the number of comments and examined the 20 issues with the most comments. This sample constituted $\geq$ 10% of their issues. For the zoo repositories from two major DL frameworks, `tensorflow/models` and `pytorch/vision`, the most-commented issues were API change requests, not defects. Here, we randomly sampled 10% of issues that met the first two filters.

Another critical aspect of our data collection was to distinguish between reengineering and non-reengineering defects. The criterion for this differentiation was based on whether the defect occurred during the process of reusing, replicating, adapting, or enhancing an existing deep learning model (*reengineering defects*) or not (*non-reengineering defects*). In other words, a reengineering defect directly related to the reengineering process and hindered the correct function or performance of the model. For instance, a defect was considered as a reengineering defect if it pertains to a problem such as the model producing incorrect results when trained with a new dataset or the model failing to perform as expected after being adapted to a new use case. Lastly, there are also some issues classified as *non-defects* that we excluded from our study. Examples include development questions or feature requests.

[3] Calculated using https://www.surveysystem.com/sscalc.html.

[4] More details can be found in §5.1.4.

These issues, although important in the broader software development process, are not directly related to the reengineering process and hence were not included in our study. According to these definitions, Table 3 provides examples of each kind of defect. From these samples, the most experienced researcher manually filtered out 93 non-defect issues, *e.g.,* development questions and feature requests. *78% (334/427* of the sampled issues included a reengineering defect.

5.1.4 Defect Analysis

Our issue classification and labeling process build on prior studies of DL defects (Islam et al. 2019; Humbatova et al. 2020). The rest of this section outlines the development of the original instrument from a pilot study, the final instrument, and data collection details. Throughout the issue analysis process, we monitored data quality using Cohen's Kappa measure for inter-rater agreement (Cohen 1960; McHugh 2012).

Instrument Development and Refinement We began by drafting an analysis instrument based on taxonomies from prior work. Previous studies have proposed taxonomies to describe DL defects, with respect to their general programming defect types, DL-specific defect types, and root causes. As a starting point, we included all the defect categories from prior DL taxonomies, recognizing that these could all theoretically occur in the reengineering process (Islam et al. 2019; Zhang et al. 2018; Humbatova et al. 2020; Thung et al. 2012; Seaman et al. 2008). By using existing taxonomies, we can compare the distribution we observed against prior work (§7.2) (Amusuo et al. 2022). To provide more insights from a process view, we enhanced the existing taxonomies to distinguish between four DL engineering components (which may be developed and validated in different phases of a reengineering process). These components are: environment, data pipeline, modeling, and training (Amershi et al. 2019; MLO 2021). More details are described in Tables 5, 6 and 7).

We refined our instrument through (Table 8) two rounds of pilot studies. Our goals were to improve inter-rater reliability and to assess the completeness of the instrument. In each round, three pairs of undergraduate analysts analyzed the DL reengineering issues from 5 repositories identified in §5.1.3, supervised by a graduate student.

- In the first round (5 repositories, 78 defects), we made several changes to the instrument as we analyzed the first three repositories, but these tailed off in the fourth repository and we did not need to make any changes in the fifth repository. After this, we concluded that our taxonomy had reached saturation (*i.e.,* covered the range of DL reengineering defects observed), and finalized the instrument. The pilot study confirmed that DL reengineering defects are a subset of DL defects (as anticipated by Fig. 1) and that existing taxonomies can categorize DL reengineering defects. After the first round, we found the taxonomy had saturated but the inter-rater reliability was low. The Cohen's Kappa value was 0.46 ("moderate").
- In the second round (5 more repositories, 53 defects), our focus was on revising the phrasing of the instrument to increase reliability. Modest changes were made to phrasing, although no new categories were introduced.

Final Instrument Our final instrument is shown in Table 5 (defect symptoms), Table 6 (general programming defect types), and Table 7 (DL-specific defect types). Changes made during data collection are indicated (see table captions for details). In our results, we mark the categories with fewer than five observed defects as *other* for clarity.[5]

Table 5 Taxonomy for defect symptoms

Defect Symptoms Category	Description
Speed Below Expectations	The code runs but the training/inference time does not match the expectation.
Accuracy Below Expectations	The code runs but the evaluation results do not match the expected accuracy.
Numerical Error	The results are Inf, NaN or Zero which are caused by division (*i.e.,* division by zero returns not-a-number value), logarithm (*i.e.,* logarithm of zero returns -∞ that could be transformed into not-a-number); Or the results appear random for each running; Or floating point overflow.
Crash	The system stops unexpectedly.
Data Corruption	The data is corrupted as it flows through the model and causes unexpected outputs.
Hang	The system ceases to respond to inputs.
Incorrect functionality	The system behaves in an unexpected way without any runtime or compile-time error/warning.
Memory exhaustion	The system halts due to unavailability of the memory resources. This can be caused by, either the wrong model structure or not having enough computing resources to train a particular model.
Other	Other symptoms that do not fall into one of the above categories.

The symptoms were adapted from Islam et al. (2019) by distinguishing two types of *Bad Performance*: *Accuracy/Speed Below Expectations*, referring to the symptoms defined by Zhang et al. (2018). The *expectations* can be different from the documentation if the code or data change. We also added *Numerical Errors* based on Wardat et al. (2021). **Bold**: Changed categories

Data Collection After refining the instrument through the pilot study, the two seniormost analysts used the instrument to analyze the data. They calibrated with one another by re-analyzing issues from the first 5 repositories studied in the pilot, and measured inter-rater agreement of 0.79 ("substantial agreement"). They then independently labeled half of the remaining issues, with a Kappa value before resolution of 0.70 ("substantial agreement"). They met to resolve disagreements.

Based on this level of agreement, and due to scheduling constraints, the remaining 132 issues were analyzed by just one of these senior analysts.[6] When needed, that analyst resolved uncertainty with another analyst.

Data Analysis To address each research question, we employ specific metrics and data analysis techniques:

– **RQ1-RQ3**: We quantified defects by categorizing them based on the specific research question being addressed (§6.1–§6.3). The primary metric used is the frequency distribution of defects, which allows us to systematically identify and analyze prevalent issues within each category. This distribution provides a clear view of the common and rare defects, offering insights into both typical and atypical issues in the reengineering process.

[5] We use *low frequency categories* to represent these categories in Figs. 8, 9 and 10.

[6] That analyst had supervised the pilot study and analyzed half of the data already, so the expected effect is consistent labels (due to his experience) but perhaps some bias in the remaining data (due to relying primarily on his perspective). The accompanying artifact indicates which issues were examined solely by this analyst.

Table 6 Taxonomy for general programming defect types

Generam Programming Defect Type Category	Description
Algorithm/method	An error in the sequence or set of steps used to solve a particular problem or computation, including mistakes in computations, incorrect implementation of algorithms, or calls to an inappropriate function for the algorithm being implemented.
Assignment/Initialization	A variable or data item that is assigned a value incorrectly or is not initialized properly or where the initialization scenario is mishandled (*e.g.,* incorrect publish or subscribe, incorrect opening of file).
Checking	Inadequate checking for potential defect conditions, or an inappropriate response is specified for defect conditions.
Data	Defects in specifying or manipulating data items, incorrectly defined data structure, pointer or memory allocation errors, or incorrect type conversions.(*i.e.,* Array, Linked List, Stack, Queue, Trees, Graphs)
External Interface	Defects in the user interface (including usability problems) or the interfaces with other systems. (e.g. API defects)
Internal Interface	Defects in the interfaces between system components, including mismatched calling sequences and incorrect opening, reading, writing, or closing of files and databases.
Logic	Incorrect logical conditions, including incorrect blocks, incorrect boundary conditions being applied, or incorrect expression.
Timing/optimization	Errors that will cause timing or performance problems.
Non-functional Defects	Includes non-compliance with standards, failure to meet non-functional requirements such as portability and performance constraints, and lack of clarity of the design or code to the reader.
Configuration (Thung et al. 2012)	Defects in non-code (*e.g.,* configuration files) that affects functionality.
Other	Other defects that do not fall into one of the above categories.

We reused a taxonomy of general programming defects from Seaman et al. (2008), adding the "Configuration" category from Thung et al. (2012). For convenience, this table presents the combined taxonomy. No new categories were added during our pilot study

– **RQ4**: To answer this question, we analyze the frequency distribution of DL-specific defect types. To gain deeper insights into the prevalent symptoms associated with each defect type, we also use a Sankey diagram. This method illustrates the relationships between DL-specific defect types and their corresponding symptoms (§6.4.2). From it, we can learn the flow and interconnections among various defect types and symptoms, highlighting the distribution and prevalence of each defect type and symptom. The results show both common and infrequent associations, suggesting areas for prioritization and de-prioritization for further study.

These methods ensure a comprehensive and clear analysis of the data, aligning with our objective to uncover both positive and negative evidence that will inform the DL reengineering process.

Table 7 Taxonomy for DL-specific defect types

DL Stage	DL-specific Defect Type Category	Description
Data pipeline	**Data preprocessing**	If an input to the deep learning software is not properly formatted, cleaned, well before supplying it to the deep learning model.
	Corrupt data (data flow bug)	Due to the type or shape mismatch of input data after it has been fed to the DL model.
	Training data quality	Due to the complexity of the data and the need for manual effort to ensure a high quality of training data (*e.g.,* to label and clean the data, to remove the outliers).
Modeling	Activation function	Incorrectly selecting the activation function of neurons.
	Layer properties	Some layer's incorrect inner properties (*e.g.,* input/output shape, input sample size, number of neurons in it).
	Missing/Redundant/Wrong layer	Adding, removing or changing the type of a specific layer was needed to remedy the low accuracy of a network.
Training	Optimizer	The selection of an unsuitable optimization function for model training.
	Loss function	Wrong selection and calculation of the loss function.
	Evaluation	Problems caused by testing and validation (*e.g.,* bad choice of performance metrics)
	Hyper-parameters	Incorrectly tuning the hyperparameters (*e.g.,* learning rate, batch size, number of epochs) of a DL model.
	Training configuration	Wrong training scripts.
	Other training process	Other faults in the training process which do not fall into one of the above categories (*e.g.,* wrong management of memory resources, wrong post-processing of the output)
Environment	API defect	Caused by APIs, this includes API mismatch, API misuse, API change, *etc.*
	GPU Usage bug	Wrong usage of GPU devices while working with DL (*e.g.,* wrong reference to GPU device, failed parallelism, incorrect state sharing between sub-processes, faulty transfer of data to a GPU device).
	Wrong environment configuration	Incorrect setting of other configurations (*e.g.,* wrong operating systems, internal interface defects).
Other	**Insufficient/Incorrect documentation**	Engineers misunderstood the documentation or they cannot find correct or sufficient instructions.

We adapted the taxonomy from Humbatova et al. (2020) by reorganizing the categories into four DL stages. We distinguish the categories of data preprocessing and corrupt data (data flow bug) based on Islam et al. (2019). **Bold**: changed/new categories

Table 8 Participant demographics

ID	SE skill	DL skill	Domain
P1	Expert	Intermediate	CV
P2	Expert	Expert	CV, NLP
P3	Beginner	Intermediate	Audio, CV, NLP
P4	Expert	Expert	CV, RL, NLP
P5	Expert	Intermediate	CV, NLP, RL
P6	Expert	Intermediate	CV, RL

P1 and P2 are open-source contributors, P3-P6 are practitioners recruited from social media platforms. Most participants self-reported here intermediate or expert skills in deep learning (DL) and software engineering (SE). All participants have experience in reengineering Computer Vision (CV) models. Some of them have also applied deep learning to Natural Language Processing (NLP), Reinforcement Learning (RL), and Audio. Note that the subjects have DL experience beyond CV; we did not observe major differences across their responses

5.2 RQ5: Interview Study on Computer Vision Reengineering

To enrich our understanding of DL reengineering challenges and practices, we triangulated the failure analysis with a qualitative data source: interviews with engineers. Our population of interest was engineers who are involved in reengineering activities in the context of computer vision. We followed the standard recruiting methods of contacting open-source contributors and advertising on social media platforms. We also complemented that data with interviews of a DL reengineering team composed of Purdue University undergraduate students with corporate sponsorship (cf. §5.2.2).

5.2.1 External Subjects: Open-source Contributors and Social Media Recruits

We recruited participants from open-source model contributors who had contributed to the projects we studied in §5.1. Our recruitment process started by sending emails to the primary contributors of each repository in Table 4. Due to a low response rate we subsequently expanded our recruiting to popular web platforms, namely Hacker News[7] and Reddit[8]. Out of the 25 open-source contributors we contacted, we received responses from 2 of them, giving us a response rate of 8%. We also received 7 responses from engineers via social media platforms.

5.2.2 Internal Subjects: Purdue's Computer Vision Reengineering Team

Inspired by the method of measuring software experiences in the software engineering laboratory described by Valett and McGarry (1989), our lab organized a DL reengineering team. This team worked to replicate research prototypes in computer vision. The goal of this team is to provide high-quality implementations of state-of-the-art DL models, porting these models from research prototypes (implemented in the PyTorch framework) to Google's TensorFlow framework. Most team members are third- and fourth-year undergraduate students. Their work was supervised and approved by Google engineers over weekly sync-ups. Our study also collect the notes from weekly team meetings as one of our data source.

[7] See https://news.ycombinator.com/.

[8] See https://www.reddit.com/.

We recognize that the use of engineering students as a data source may seem unsound. Here we discuss some reasons why this data source may be relevant to industry. We specifically interviewed the team leaders — each of the team's projects had 1-2 student leaders. The team leaders were fourth-year undergraduates, sometimes working for pay and sometimes for their senior thesis. All team leaders contributed to at least two reengineering projects. All team members received team-specific training via our team's 6-week onboarding course on DL reengineering. Team leaders typically worked for 15-20 hours per week over their 2 years on the team. Ultimately, the proof is in the pudding: the team's industry sponsor (Google) uses the team's replication of 7 models in production in their ML applications, and has published them open-source in one of the most popular zoo repository we studied in §5.1, the TensorFlow Model Garden.

The completed models required 7,643 lines of code, measured using the cloc tool (AlDanial 2022). The estimated cost of the reengineering team's work was ∼$105K ($15K/model): $40K on wages and $65K on computation ($5K for VM and storage, $60K for hardware accelerator rental [e.g., GPU, TPU]).

5.2.3 Interview Data Collection

To design our interview, we followed the guideline of *framework analysis* which is flexible during the analysis process (Srivastava and Thomson 2009). A typical framework analysis include five steps: data familiarization, framework identification, indexing, charting, and mapping (Ritchie and Spencer 2002). Our interview follows a three-step process modeled on the *framework analysis* methodology:

Data Familiarization and Framework Identification We created a *thematic framework* based on the reengineering challenges and practices we identified from our literature review (§2) , as well as our findings from the open-source failure analysis (§5.1). This framework includes the challenges and practices of bug identification and testing/debugging. We also created a draft reengineering workflow for reference in our exploration of reengineering practices.

Interview Design: We designed a semi-structured interview protocol with questions that follow our identified themes of DL reengineering. We conducted two pilot interviews to refine our framework and interview protocol. The final interview protocol includes four main parts: demographic questions, reengineering process workflow, reengineering challenges, and reengineering practices. Table 9 indicates the three main themes of our interview protocol. Our research team developed an initial understanding of the challenges and practices of DL reengineering from the failure analysis study. This experience informed the design of the interview, especially some of our follow-up questions. For example, Q6 and Q7 on DL stages, Q14 on DL model documentation, and Q16 on the acceptable trade-off between model performance and engineering cost, were all shaped by the findings from our failure analysis study. These interviews were technical and focused, more closely resembling a structured interview than a semi-structured one. During the interview, we provided relevant themes and showed the draft of the reengineering workflow in a slide deck.[9]

Filtering Interviews for Relevance: We applied three inclusion criteria for the interview participants recruited from the social media platforms: (1) the participant should have industry experience, (2) hold at least a bachelor's degree, and (3) have experience in CV and/or DL

[9] The final version of the reengineering workflow is shown in Fig. 12. The colored annotations on that figure were not shown to the subjects.

Table 9 Interview protocol addressing RQ5

Themes	Questions
Process	**Q1:** *Can you talk me through the process that your team follows to re-engineer a machine learning model from research paper/existing implementation/another engineer's project?*
	Q2: *Please take a look at our draft workflow. Can you tell me if you think this is an accurate process workflow?*
	Q3: *Would you like to add any back-edges in this diagram?*
	Q4: How does your team update new iterations of your model if it doesn't work for the first time?
Challenges	**Q5:** *Which parts do you think are challenging when re-engineering a model*
	Q6: Can you tell me about an error you found in TRAINING/MODELING/DATA PIPELINE?
	Q7: Can you describe any challenges you met when implementing TRAINING/MODELING/DATA PIPELINE?
	Q8: How do you address these challenges?
	Q9: Have you met any challenges when integrating all components?
	Q10: Can you think about 1-2 changes to the reengineering process that would make this process easier for you?
Practices	**Q11:** *How does your team work together to make the process more effective?*
	Q12: How do you decide an existing implementation is trustworthy?
	Q13: What do you find is helpful/problematic in a DL research paper?
	Q14: What do you find is helpful/problematic in the documentation of DL models?
	Q15: Are there existing tools (or other technologies) you found valuable/problematic for re-engineering?
	Q16: How do you determine the acceptable trade-off between the performance of the model (accuracy/speed) and the cost of your team (time/money, etc.)?

We answer RQ5 by combining results from four kinds of questions: demographic questions, process workflow, challenges, and effective practices. Details of demographic questions can be found in our artifact (§10). The interview is semi-structured. The questions in italics are questions that all participants were asked. The other questions are examples of follow-up questions

reengineering. Four of the seven subjects from the social media platform group met these requirements. The subjects recruited from open-source contributors and our CV reengineering team had relevant experience.

Overall, our qualitative data comprised 6 external interviews (2 open-source contributors, 4 industry practitioners) and 6 internal interviews (leaders of CV reengineering team after the team had been operating for 18 months). Table 8 summarizes the demographic data of the 6 external participants.

5.2.4 Data Analysis

The interview recordings were transcribed by a third-party service.[10] Themes were extracted and mapped by one researcher, the same person who conducted the interviews. We compared those parts of each transcript side-by-side, and noted challenges or practices that were

[10] See https://www.rev.com/. One of the interviews was conducted in Chinese by a researcher who is a native speaker; that researcher listened to the recording and refined the transcript.

discussed by multiple leaders and extracted illustrative quotes. By analyzing the interview transcripts, we were able to understand the larger picture and summarize common challenges. After completing the interviews, we performed member checking (Birt et al. 2016) with the 6 CV reengineering team leaders to validate our findings. They agreed with our analysis and interpretations.

6 Results and Analysis

6.1 RQ1:What DL reengineering defect manifestations are more common, in terms of project type, reporter type, and DL stages?

Finding 1: *Project types:* User "forks" contain the most basic (73%) (Fig. 5). *Reporter types:* Most defects (58%) are reported by re-users. Defects reported by replicators are rarely identified (<1%). Almost all reproducibility defects are reported by re-users (Fig. 6). *DL stages:* 91% defects are reported during environment, training, and data pipeline. However, 68% of reproducibility defects occur in the training stage (Fig. 7).

To understand the characteristics of DL model reengineering, we analyzed the distribution of reengineering phases in terms of reporter types and DL stages.

Project types: Figure 5 shows that most defects are basic defects (73%) located in user "forks". Most of the basic defects in user "forks" are due to the misunderstanding of the implementation, miscommunication, or insufficient documentation of the model(s) they are using. In the discussions, we saw the owners of the repositories often tell the re-users to

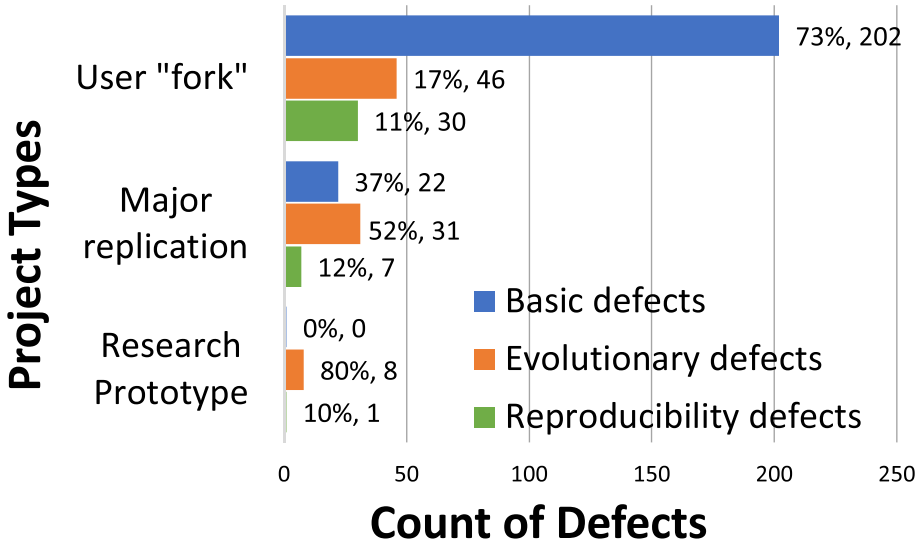


Fig. 5 Reengineering phases in different project types. Most defects (80%, 278/348) are identified in user "forks", among which 73% (202/278) are basic defects. In this and subsequent figures, the indicated percentages are calculated relative to the corresponding type

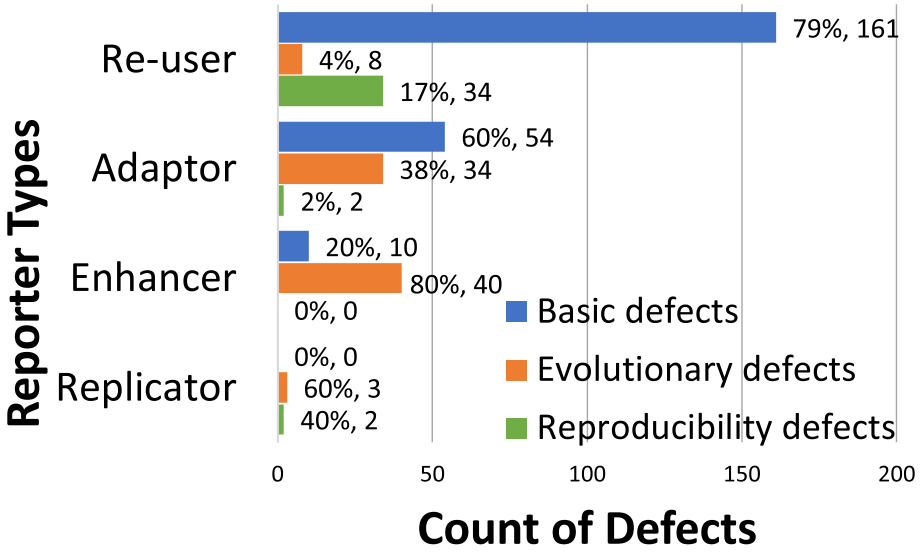


Fig. 6 Reengineering phases vs. Reporter types. 58% (203/348) of the defects are reported by re-users. Less than 1% are reported by replicators. Almost all reproducibility defects (89%, 34/38) are reported by re-users

read the documentation, while the re-users would remark that the documentation is confusing. This observation supports the suggestions from Gundersen *et al.* on documentation improvement (Gundersen and Kjensmo 2018) and indicates there is still a need for detailed documentation and tutorials, especially for the reengineering process (Pineau 2022).

Reporter types: Figure 6 indicates the distribution of reengineering phases based on the reporter type. Most defects are reported by re-users, followed by adaptors and then enhancers. Something to note here is that almost all reproducibility defects are reported by re-users. This

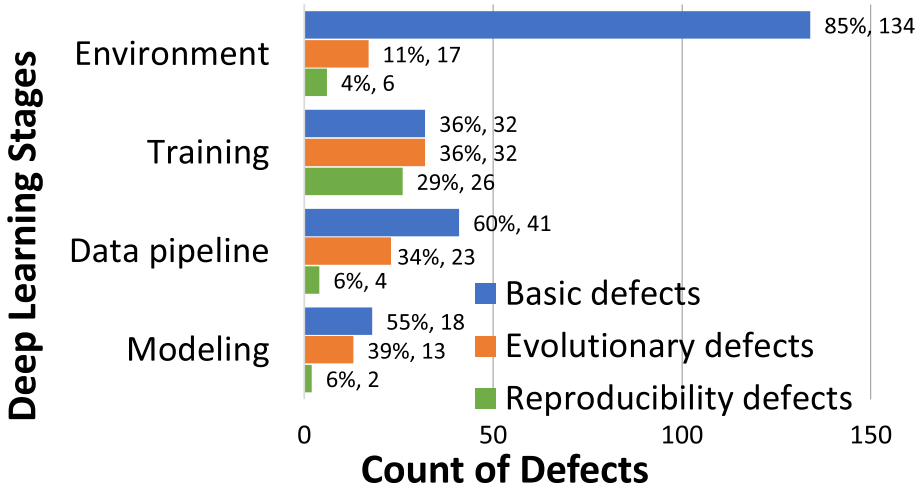


Fig. 7 Reengineering phases by DL stage. Most *reproducibility* (68%, 26/38) and *evolutionary* (38%, 32/85) defects occur in Training stage. Data pipeline also has many *evolutionary* (34%) defects

finding somewhat follows from our reporter type definitions — an adaptor uses a different dataset, and an enhancer adds new features to the model, so neither seems likely to report a reproducibility defect. However, the absence of reproducibility defects from replicators was more surprising. Perhaps replicators are more experienced, and more likely to fix the problem themselves than to open an issue. Alternatively, perhaps there are simply far more re-users in this community.

DL stages: Figure 7 shows the distribution of reengineering phases by deep learning stage. We observe that the modeling stage results in the fewest reported defects. Only 9% (33/348) of defects are reported in the modeling stage. The plurality of defects (45% or 157/348) occur in the environment, followed by 26% (90/348) of defects in the training stage and 20% (68/348) in the data pipeline stage.

By type, 60% of defects in the data pipeline and 55% of defects in the modeling stages are basic defects, which means that they are easy to identify. This kind of reengineering phase can be identified from error messages or visualization of the input data. In contrast, the majority (68%) of reproducibility defects we found were concentrated in the training stage. Evolutionary defects occurred in each DL stage, with many (38%) manifesting in the training stage.

The data indicates that the training stage introduces the most challenging debugging in the reengineering process. Most of the training defects do not result in crashes, but lead to the mismatches between the reimplementation and specification/documented performances. A possible reason is that replicators may have less available data about training — they can refer to the existing replications/prototypes to reuse the data pipeline or the same architecture (*e.g.,* backbones) when dealing with similar tasks, but training records may not be available (Chen et al. 2022a).

6.2 RQ2: What types of general programming defects are more frequent during Deep Learning reengineering?

Finding 2: Most Environment defects are *interface* defects (88%). The Data Pipeline and Modeling stages have similar distributions oriented towards *assignment/initialization* defects. Training defects are diverse. (Figure 8)

Here we consider the defect types by DL stage (Fig. 8). 88% of the defects in the environment configuration are interface defects. These defects can be divided into *external* interface defects (62%, 98/157), *i.e.,* the user interface or the interfaces with other systems; and *internal* interface defects (25%, 40/157), *i.e.,* the interface between different system components, including files and datasets (Seaman et al. 2008). For the external environment, engineers have to configure the DL APIs and hardware correctly before running the model. However, there are often inconsistencies between different DL frameworks and different hardware, leading to defects. For the internal environment, engineers need to set up the modules and configure the internal paths, but the documentation or tutorials of the model are sometimes incomprehensible and result in defects.

During training there are relatively more algorithm/method and timing/optimization defects, compared to other stages. Engineers appear prone to make mistakes in algorithms and methods when adapting the model to their own datasets, or to fail to optimize the model in the training stage.

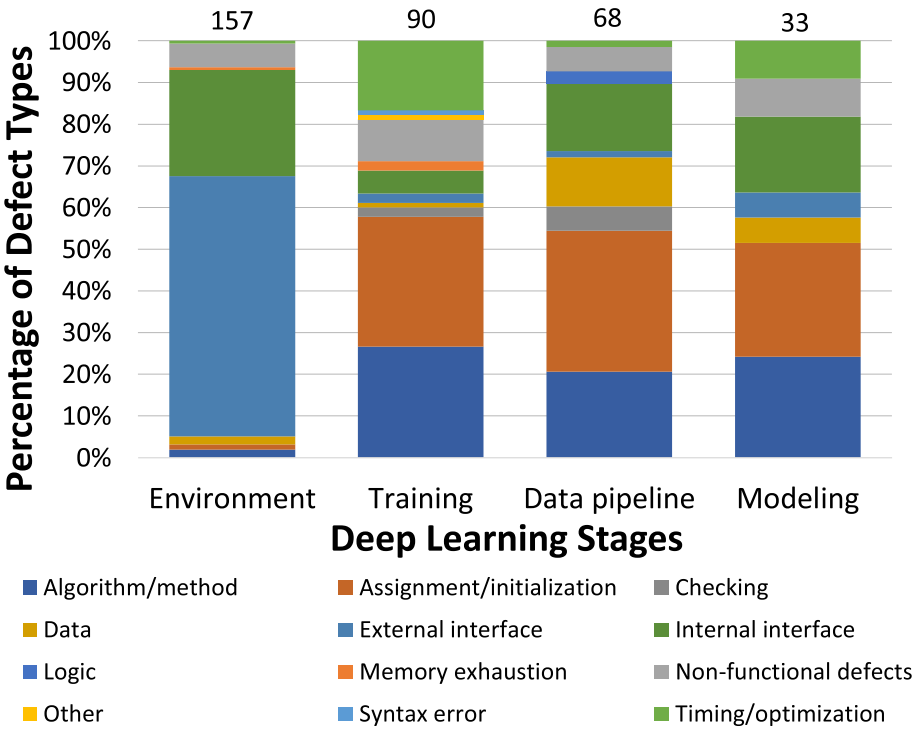


Fig. 8 Defect types by DL stage. Most Environment defects are *interface* defects. The Data Pipeline and Modeling stages have similar distributions oriented towards *assignment/initialization* defects. Training defects are diverse

We observe that assignment/initialization defects account for 34% (23/68) in data pipeline stage and 27% (9/33) in the modeling stage. Internal interface defects also account for 16% (11/68) of the defects in the data pipeline. To fix assignment/initialization defects and internal interface problems, engineers only need to change the values or set up the modules and paths correctly. These are relatively simple defects and simple tool support could help.

6.3 RQ3: What are the most common symptoms of Deep Learning reengineering defects and how often do they occur?

Finding 3: Across the environment, data pipeline, and modeling stages, the most frequent symptom is *crash* (62-83%). Training appears to be the most challenging stage, where *Accuracy below expectations* accounts for the largest proportion (34%) of defects. (Figure 9)

Figure 9 shows the distribution of defect symptoms in different DL stages. Our data shows that *Crash* is a common symptom. 83% (130/157) defects result in crashes in environment, data pipeline, and modeling. Most crashes happen due to incorrect environment configuration, *e.g.,* internal interface, APIs, and hardware.

In contrast, 72% (65/90) of defects in the training stage do not result in crashes (34% lead to *Accuracy below expectations*). The training defects are more likely to result in lower

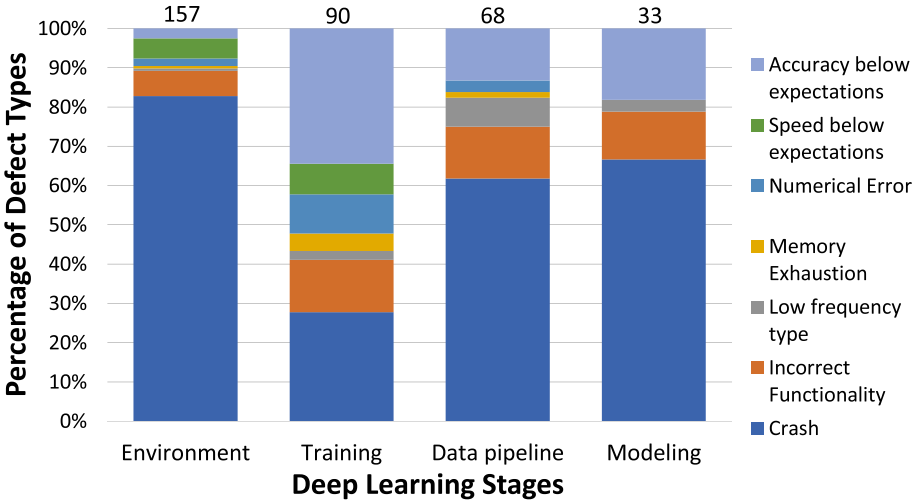


Fig. 9 Defect symptoms (Islam et al. 2019; Zhang et al. 2018) by DL stage. Across most stages, the most frequent symptom is *crash* (62-82%). Meanwhile, in Training there are many symptoms, with *Accuracy below expectations* accounting for the largest proportion (34%) of defects

accuracy and incorrect functionality which are harder to identify. Locating the defect could be more time-consuming because the fixers have to train the model for many iterations and compare the accuracy or other metrics to see whether the training works properly. Based on this, we believe that training is the most challenging stage.

Similar to the distribution shown in Figs. 7 and 9 indicates that reproducibility and evolutionary defects (*e.g.,* lower accuracy and incorrect functionality) can be located in any of the four stages. Reproducibility defects are hard to debug, especially when the reimplementation is built from scratch. For evolutionary defects, since the code or data has been changed based on the research prototypes or replications, the defects are more likely to be identified in the changed parts which can be easily found.

6.4 RQ4: What are the most common Deep Learning-specific reengineering defect types?

Finding 4: Most Environment defects are caused by API defects (46%, 73/157). In the Data Pipeline, *data preprocessing* defects predominate (38%, 26/68). Most Modeling defects are due to *model/weight operations* (67%, 22/33). Training defects have diverse causes. (Figure 10)
Finding 5: In DL reengineering, a given defect symptom can result from many distinct DL-specific defect types (Fig. 11). That makes it challenging to diagnose a defect.

6.4.1 Distribution of DL-specific Defect Types

To answer RQ4, we analyzed the distributions of defect root causes (Fig. 10) and present our findings by DL stage. In our study, all DL stages are determined based on the location where the fix was made in.

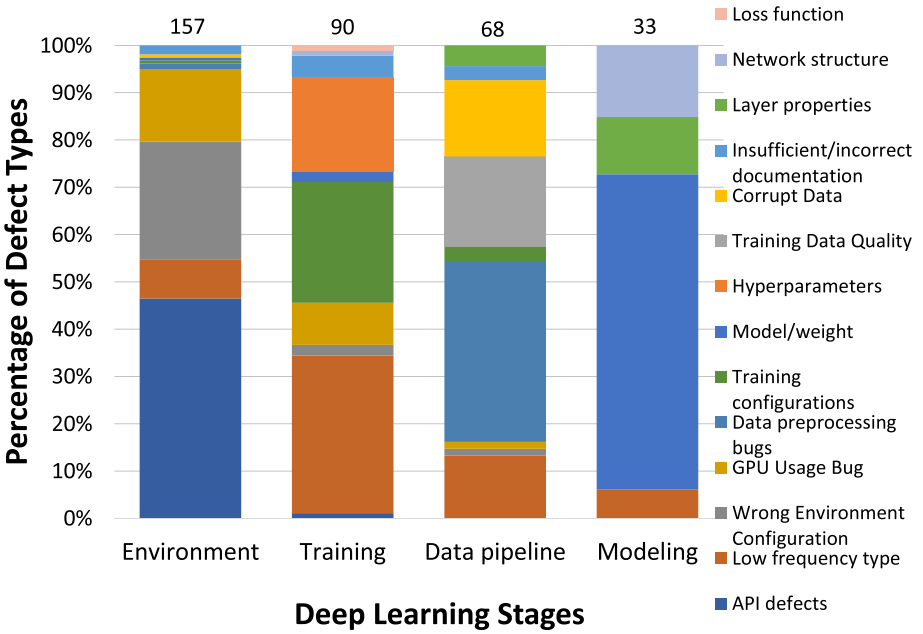


Fig. 10 DL-specific defect types by stage. In the Data Pipeline, *data preprocessing*, *corrupt data* and *data quality* predominate. Most Modeling defects are due to *model/weight operations* and *network structure*. Training defects have diverse causes

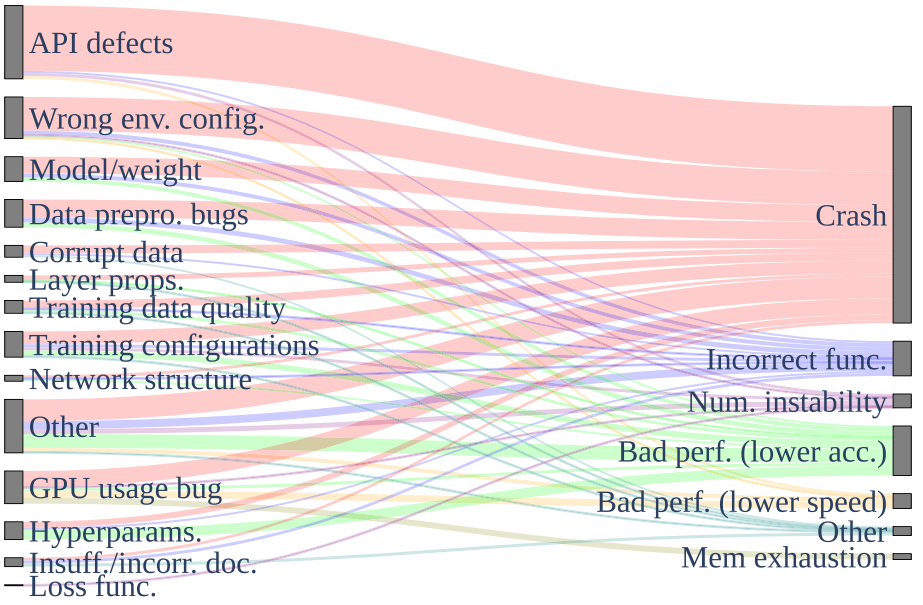


Fig. 11 Sankey Diagram showing the relationship between DL-specific reengineering defect types (left side) and symptoms (right side). DL reengineering defects are hard to diagnose because of the many distinct root causes for most of the observed symptoms

Environment Figure 10 shows that most of the environment defects are caused by API defects (46%, 73/157), Wrong environment configuration (25%, 39/157), and GPU usage defects (15%, 24/157). Many reusers reported defects due to API changes and mismatches. Insufficient documentation can easily lead to misunderstandings and confusion. The portability of models is another problem, especially for the GPU configuration.

Data pipeline Figure 10 indicates three main root causes in the data pipeline: data preprocessing (38%, 26/68), training data quality (19%, 13/68), and corrupt data (16%, 11/68). Engineers are likely to have troubles in data processing and data quality. These defect types are especially frequent for adaptors who are using their own datasets. Datasets vary in format and quality compared to the benchmark CV datasets (*e.g.,* ImageNet (Krizhevsky et al. 2012), COCO (Lin et al. 2014)). Therefore, before adaptors feed the data into the model, they have to first address the dataset format, shape, annotations, and labels. Moreover, customized datasets are less likely to have enough data to ensure a comparable level of accuracy, so it is necessary to use data augmentation in their data pipeline. However, as we observed, some engineers did not realize the significance of data augmentation and data quality in their reengineering tasks which lead to the lower accuracy.

Modeling The main root causes in the modeling stage are *Model/weight* (67%, 22/33), layer properties (12%, 4/33), and network structure (9%, 5/33). These causes represent the incorrect initialization, loading, or saving of models and weights. We observed that some reengineering work involves moving a model from one DL framework to another. Though some tools exist for the interoperability of models between DL frameworks (Liu et al. 2020; Jajal et al. 2023), we did not see them in use, and engineers are still having troubles in the modeling.

Training There are multiple defect types contributing to training defects. The top two are training configurations and hyper-parameters. Training configurations include different training algorithms (*e.g.,* non-maximum suppression, anchor processing) and some specific parameters which are used to configure the training, but different from the hyper parameters. When engineers have different environment configurations or adapt the model to their own datasets, it is necessary to modify the training algorithms and tune the hyper parameters in a proper way so that the training results match their expectations.

6.4.2 Relationship between DL-specific Defect Types and Symptoms

Identifying defects during DL reengineering is challenging due to the complex interplay between DL-specific defect types and manifest symptoms. Figure 11 maps the relationships between DL reengineering defect DL-specific defect types and symptoms. As mentioned in §5.1.3, we treat defects originating from various DL-specific defect types as separate entities, thereby ensuring that each symptom is attributed to a singular cause. The three most common symptom types — crash, low accuracy, and incorrect functionality — all have many possible DL-specific defect types. Most system crashes stem from API defects, incorrect environment configurations, and GPU usage bugs. Performance issues are spread across several causes, including training configurations, hyperparameters, and many other causes in low frequent types. Incorrect functionality emerges from a variety of causes across multiple stages of the DL stages.

6.5 RQ5: When software engineers perform Deep Learning reengineering, what are their practices and challenges?

Finding 6: *Challenges*: The four main challenges in the DL reengineering process are model operationalization, performance debugging, portability of DL operations, and customized data pipeline. *Practices*: (Interface): Interface can help unify and automate model testing, especially testing of component integration. (Testing): Validation was a common emphasis. Team employed complementary testing techniques (*i.e.,* unit, differential, and visual testing). (Debugging): Comparing evaluation metrics after just 25% of training is a shortcut.

Our failure analysis shed some light on DL reengineering challenges, but little on the guidelines for ML reengineering requested in prior works (Rahman et al. 2019; Devanbu et al. 2020). In this section, we describe those aspects based on the experiences of our CV reengineering team. First, we describe three main reengineering challenges we found in our reengineering team. Then, in §6.5.2 we describe three practices we identified from interview study to mitigate those challenges, including interface, testing, and debugging.

6.5.1 Reengineering Challenges

We interviewed 2 open-source contributors, 4 industry practitioners, and 6 leaders of the team (§5.2). Our interview study identified 8 reengineering challenges within the process (Table 10). We focus on the four challenges that were mentioned by at least three participants.

Challenge 1: Model Operationalization

> ***Practitioner 1****: "The only way to validate that your model is correct is to...train it, but the pitfall is that many people only write the code for the model without any training code...That, in my opinion, is the biggest pitfall."*
>
> ***Practitioner 9****: "I would say that understanding the previous work is often the most difficult part ...translating that information and transmitting that information in an efficient way can be very hard."*
>
> ***Leader 2****: "So one challenge...digging through their code and figuring out what is being used and what isn't being used."*
>
> ***Leader 3****: "And in the paper, they...give a pseudo code for just iterating through all..., which they mentioned in the paper that is really inefficient. And they do mention that they have an efficient version, but they never explain how they do it [in the paper]."*
>
> ***Leader 5****: "A lot of the work is figuring out what they did from just the paper and the implementation, which is not exactly clear what they're doing."*

There may be multiple implementations or configurations of the target model. The interview participants found it hard to distinguish between them, to identify the reported algorithm(s), and to evaluate their trustworthiness. First, some research prototypes are not officially published. To correctly replicate the model, it is hard to identify which implementation they could refer to. Moreover, the model may be different from the one in the paper.

Table 10 Challenges during deep learning reengineering process

Challenge	Description	# Participants	Reeng. Activities
1. Model Operationalization	Unclear implementations, configurations, or scripts of the original model.	8	Reuser, Replicator, Adaptor, Enhancer
2. Performance Debugging	Debugging and achieving expected end-to-end performance metrics.	8	Reuser, Replicator, Adaptor, Enhancer
3. Portability of DL Operations	Discrepancies in function naming or signatures across frameworks, inconsistencies in behaviors of the 'same' operations, limitations on specific hardware.	6	Reuser, Replicator, Enhancer
4. Customized data pipeline	Data processing, quality assurance, and pipeline implementation.	4	Replicator, Adaptor, Enhancer
5. Team collaboration	Inconsistent implementations among multiple team members.	2	Replicator, Enhancer
6. Code Readability	Obfuscated code.	1	Enhancer
7. Evaluation Metrics	Identifying the best metric to evaluate the model.	1	Adaptor
8. Data availability.	Unavailable training data.	1	Adaptor

The third column shows how many participants (of 12) mentioned the challenge. Relevant reengineering activities identified in interview data are also incorporated

Leader 3 reported that the research prototype used a different but more efficient algorithm without any explanation. Even for the original prototypes, the leaders reported that the prototype's model or training configuration may differ from the documentation (*e.g.,* the research paper). These aspects made it hard for the reengineering team to understand which concepts to implement and which configuration to follow.

Challenge 2: Performance Debugging

> ***Practitioner 1****: "If your data format is complicated then the code around it would also be complicated and it would eventually lead to some hard to detect bugs."*
>
> ***Practitioner 3****: "[Performance bugs] are silent bugs. You will never know what happened until it goes to deployment."*
>
> ***Leader 1****: "You can take advantage of...differential testing but then someone shows out a probabilistic process to this...it becomes really difficult because the testing is dependent upon a random process, a random generation of information."*
>
> ***Leader 4****: "Sometimes it gets difficult when the existing implementation... [doesn't] have the implementation for the baseline. So we have to figure out by ourselves. Paper also doesn't have that much information about the baseline...if you're not able to even get the baseline, then going to the next part is hard..."*

Another main challenge we observed is matching the performance metrics of existing implementations, especially the original research prototype. These metrics include inference

behavior (*e.g.,* accuracy, memory consumption) and training behavior (*e.g.,* time to train). Multiple interview participants stated that performance debugging is difficult. Even after replicating the model, performance still varies due to hardware, hyper-parameters, and algorithmic stochasticity. Compared to Amershi *et al.'s* general ML workflow (Amershi et al. 2019), during reengineering we find that engineers focus less on neural network design, and more on model analysis, operation conversions, and testing.

Challenge 3: Portability of DL Operations

> ***Practitioner 2****: "The biggest problem that we encounter now is that some methods will depend on certain versions of the software...When you modify the version, it will not be reproducible...For example, the early TensorFlow and later TensorFlow have different back propagation when calculation gradient. Later updated versions will have different results even when running with the same code...Running our current code with CUDA 11.2 and PyTorch 11.9, the program can be executable, but the training will have issues....The inconsistency between PyTorch and Numpy versions could make the results the same every time."*

> ***Leader 3****: "There was basically [a] one-to-one [mapping of] functions from DL_PLATFORM_1 [to our implementation in] DL_PLATFORM_2. But halfway through...we realized we needed to make it TPU friendly ...had to figure out a way to redesign...to work on TPU since the MODEL_COMPO NENTS in DL_PLATFORM_1 were all dynamically shaped."*

> ***Leader 2****: "They don't talk about TPUs at all...If you're in a GPU environment, it's a lot easier also, but in order to convert it to TPU, we had to put some design strategies into place and test different possible prototypes."*

> ***Leader 6****: "The most challenging part for us is the data pipeline...a lot of the data manipulation...in the original C implementation is...hard [in] Python."*

Though engineers can refer to existing implementations, the conversion of operations between different DL frameworks is still challenging (Liu et al. 2020). The interview participants described four kinds of problems. (1) Different DL frameworks may use different names or signatures for the same behavior. (2) The "same" behavior may vary between DL frameworks, *e.g.,* the implementation of a "standard" optimizer.[11] (3) Some APIs are not supported on certain hardware (*e.g.,* TPU), and the behavior must be composed from building blocks. (4) Out of memory defects could happen in some model architectures when using certain hardware (*e.g.,* TPU). We opened the issue in a major DL framework and the maintainers are investigating.[12]

Challenge 4: Customized Data Pipeline

> ***Practitioner 1****: "If your data format is complicated then the code around it would also be complicated and it would, eventually, lead to some hard to detect bugs."*

> ***Practitioner 4****: "Sometimes some data can mess the whole model prediction. We need to get them out of our project...It probably consumed 30% of our entire project time."*

[11] The reengineering team identified and disclosed three examples of this behavior to the owners of the DL framework they used. The documentation of a major DL framework was improved based on our reports.

[12] See https://github.com/tensorflow/models/issues/10528.

Practitioner 6*: "[Data pipeline] is the most challenging part because it's not like there's a formula of things that you can do...When it comes to data pipeline, it's more of an art than science"*

Leader 1*: "The data pipeline is the hardest to verify and check because testing it takes so long. You need to train the model in order to test the data pipeline."*

Customizing the data pipeline is a challenging stage in the DL reengineering process, especially when engineers want to adapt the original model to a new dataset. However, our qualitative data also suggest that this can occur during the replication and enhancement stages, where a customized data pipeline may be needed. For instance, replicating a model in a different framework requires corresponding data preprocessing. Similarly, enhancing a model to support different data formats also necessitates adjustments to the data pipeline. The interview participants described challenges in data processing, data quality, and the implementation of the data pipeline. First, the data processing is challenging because there is no standardized solution. The practitioners, especially model adaptors, needs to process the data properly to match the original model input. Second, the data quality can affect the model performance a lot. For example, Practitioner 4 mentioned that some data can mess the whole model prediction and it is time consuming to address these issues. Third, the data pipeline can be very different when using different programming languages and DL frameworks. Implementation and testing can be time-consuming.

6.5.2 Reengineering Practices

We also summarized three reengineering practices based on our interview analysis:

Practice 1: Starting with the Interface

Practitioner 2*: "We will define the API for the [our own] interface first. In such case, if something goes wrong during the component integration, we will easily localize the defect."*

Practitioner 3*: "We look for implementations which are very quick to start with, which have a very good API to start."*

Leader 4*: "If you're using existing implementations, you should take time to familiarize with the existing things [code base] that we already have."*

Leader 6*: "The DL_FRAMEWORK interface, for the MODEL_ZOO specifically, I think that's very important."*

Interface can help unify and automate model testing. Typically, an existing code base has a unified structure which includes the basic trainer and data loader of a model. The interface is essential if the reengineering team is required to use existing code base or model zoos, such as Banna *et al.* describe for the TensorFlow Model Garden (Banna et al. 2021). *Practitioner 2* indicated that implementing interface APIs first can help a lot on bug localization during component integration. Their team also combine agile method (Cohen et al. 2004) with the use of interface APIs which makes the team collaboration more effective. Our reengineering team shares similar experience by implementing interface API first. *Leader 4* indicated that

team members have to get familiar with the code base and relevant interface APIs first before the real model implementation.

Practice 2: Testing

> ***Leader 4****: "Unit testing was eas[ier]....because if you're just trying to check what you're writing, it is easier than always trying to differentiate your test and match it with some other model."*

> ***Leader 1****: "You can load the...weights from the original paper. [It] might be a little bit difficult to do but you could do that [and] test the entire model by...comparing the outputs of each layer."'*

Based on the interviews and notes from weekly team meetings, we found that complementary testing approaches were helpful. We describe the team's three test methods: unit testing, differential testing, and visual testing.

Unit testing can ensure that each component works as intended. For the modeling stage, a pass-through test for tensor shapes should be done first, to check whether the output shape of each layer matches the input shape of the next. They also do a gradient test which calculates the model's gradient to ensure the output is differentiable.

Differential testing compares two supposedly-identical systems by checking whether they are input-output compatible (Mckeeman 1998). This is most applicable in the Modeling stage: the original weights/checkpoint can be loaded into the network, and the behavior of original and reengineered model can be compared. This technique isolates assessment of the model architecture from the stochastic training component (Pham et al. 2020).

Differential testing is also applicable in Environment and Training stages to ensure the consistency of each function. For example, when testing the loss function in the training stage, they generate random tensors, feed them into both implementations, and compare the outputs. The outputs should match, up to a rounding error.

> ***Leader 4****: "Both...implementations, even though we are doing the same model, the way they organized are very different....Differential testing can get difficult."*

> ***Leader 5****: "In the data pipeline, one of the big challenges is that the data pipeline is not deterministic, it's random. So it's hard to make test cases for it nor to see if it matches the digital limitation, because you can't do differential testing because it's random."*

However, differential testing does not apply to all DL components. For example, the data pipeline has complex probabilistic outputs, where they found it simpler to do *visual testing*. For data pipeline, each preprocessing operation can be tested by passing a sample image through the original pipeline and the reengineering pipeline. They visually inspect the output of two pipelines to identify noticeable differences. This approach is applicable in CV, but may not generalize to other DL applications.

Unit, differential, and visual testing are complementary. At coarser code granularities, automated differential tests are effective; at a fine-enough level of granularity the original and replicated component may not be comparable, and unit tests are necessary. The characteristics of a data pipeline are difficult to measure automatically, and visual testing helps here.

Practice 3: Performance Debugging

> ***Leader 1****: "I think the biggest thing is logging the accuracy after every single change...then you know what changes are hurting your accuracy and which changes are helping, which is immensely helpful."*

***Practitioner 1**: "If your data format is complicated, then the code around it would also be complicated and it would, eventually, lead to some hard to detect bugs."*

The most common way to detect reproducibility and evolutionary defects is to compare model evaluation metrics. This approach can be costly due to the resources required for training in a code-train-debug-fix cycle. However, as we noted from the team meetings, 70% of the final results may be achieved within 10-25% of training. They thus check whether evaluation metrics and trends are comparable after 25% of training, shortening the debugging cycle.

Beyond this approach, they agree with prior works that record-keeping help model training and management (Schelter et al. 2017; Vartak et al. 2016).

7 Discussion and Implications

In this section, we first triangulate our findings into a reengineering workflow (§7.1). Then we compare our findings to prior works (§7.2) and propose future directions (§7.3).

7.1 Triangulating our findings

We compare (§7.1.1) and contrast (§7.1.2) the two prongs of our case study. Then we synthesize them into a hypothesized DL reengineering workflow (§7.1.3).

7.1.1 Similar challenges (in model implementation and testing)

The two prongs of our case study identified similar challenges in *model implementation and testing*, especially *performance debugging*, which we show below in Fig. 12. Our failure analysis found that most reengineering defects are caused by DL APIs and hardware configuration (see Figs. 7 and 8), and we identified the similar challenge of *portability* from our interview study. Moreover, in the failure analysis we suggested that the training stage would be the most challenging because of the diversity of failure modes (cf. Figs. 8, 9, and 10). This stage was also highlighted by our interview participants as the *performance debugging* challenge. In addition, we observed multiple testing strategies applied by engineers in the failure analysis, and heard about some of them in more details in the interview study, including unit testing, differential testing, and visual testing (§6.5.2).

7.1.2 Divergent findings (in model operationalization)

However, our two data sources were not fully in agreement. The interviews identified the *model operationalization* challenge, while the failure analysis data provided no direct evidence of this challenge. We believe that when engineers encounter model operationalization defects, they may not open an issue about it, or may be unable to identify the defect until testing. In this case, the *operationalization* challenge could manifest as a *performance debugging* defect. This silence in the failure analysis part may be related to the low number of issues that we attributed to the "Replicator" reported type (*i.e.,* the persona most likely to encounter this kind of issue).

7.1.3 Observed DL Reengineering Workflow

To illustrate DL Reengineering challenges and practices, we developed an idealized reengineering workflow (see Fig. 12 below) based on our failure analysis and interview data. To develop this, we had one leader of the student DL reengineering team describe the reengineering workflow that their sub-team followed. The other student leaders revised it until agreement. Then we revised it based on our observations of the open-source failure data and the interview data.

The resulting workflow has three main stages: (1) Model selection and analysis; (2) Implementation and testing; and (3) Review and release. In the first stage, a DL reengineering team identifies a candidate model for the desired task (*e.g.,* low-power object recognition), and determines its suitability on the target DL framework and hardware. Existing implementations are examined as points of comparison. In the second stage, the components of the system are implemented, integrated, and evaluated, possibly with different personnel working on each component. At the end of this stage, the model performance matches the paper to a tolerance of 1-3%. In the third stage, the model is tailored for different platforms, *e.g.,* servers or mobile devices, and appropriate checkpoint weights by platform and dataset are published. We included example checklists for these tasks in the supplemental material.

The reader may note the relative linearity of the model given in Fig. 12. Our findings from both the failure analysis and the interview study suggest that during DL reengineering, the system requirements and many design elements are well understood, reducing the need for iteration. The development process can then be more plan-based rather than the iterative (agile) approach common in software development (Keshta and Morgan 2017). This finding contrasts with the DL model development and selection workflows identified in prior work (Amershi et al. 2019; Jiang et al. 2023b), indicating that different kinds of DL engineering work involve different degrees of uncertainty.

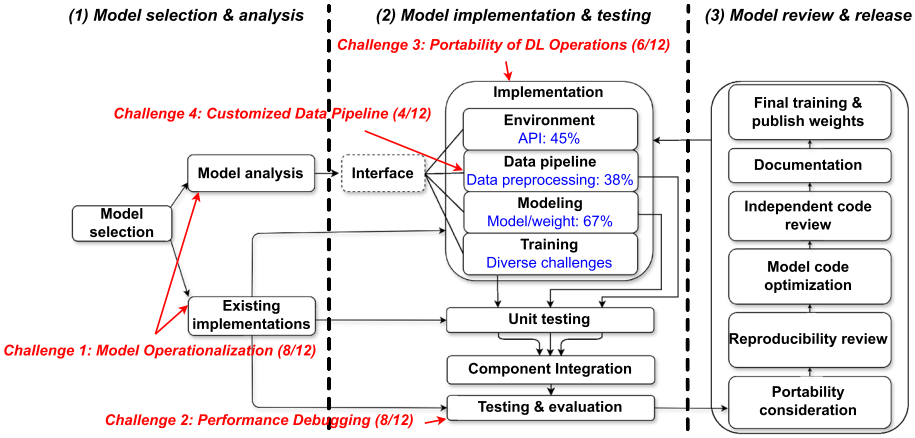


Fig. 12 Reengineering workflow, divided into three main stages. Forward edges denote the reengineering workflow. Dotted line denotes optional (interface). Back edges denote points where errors are often identified. Red arrows indicate the main location of each challenge, Blue text denotes the main root causes of failures in each DL stage (see Fig. 10)

Recall in Fig. 1, we described four sub-activities: reuse, replicate, adapt, enhance. Based on our analysis, here is how they integrate into Fig. 12:

- ***Reuse:*** The reuse process is a sub-graph of the overall process, with minimal activity needed within the second stage (*i.e.,* model implementation & testing) Reusers typically have precise requirements before selecting a model. During model reuse, significant time is dedicated to analyzing the model, integrating it, and evaluating its performance. This process mirrors the reuse dynamics discussed in prior work (Jiang et al. 2023b). Challenges include operationalizing the model and assessing its performance effectively (Table 10).
- ***Replicate:*** Our data sources were focused on the replication process. A critical component of this process is leveraging the existing implementation for testing, with differential testing being a prevalent and vital method. Challenges in replication may include issues related to the portability of DL operations (Table 10).
- ***Adapt:*** The adaptation process needs additional activities in the implementation, specifically data pipeline. During adaptation, changes in the downstream dataset (as defined in Table 1) can necessitate adjustments to the implementation. Additionally, dataset modifications often lead to challenges in developing a customized data pipeline, which may include issues like poor training data quality or corrupted data (Table 7, Fig. 10).
- ***Enhance:*** The enhancement process includes additional activities in (1) selection and analysis, and (2) implementation and testing. The enhancement usually initiates with the existing implementations and requires a comprehensive understanding of pertinent topics, such as state-of-the-art model architecture or training methods. Enhancers typically enter this phase with a clear goal (§6.5) — such as adopting a different loss function or attaining a specific performance target in a designated deployment environment — which guide both the enhancement efforts and subsequent evaluations (Figs. 6 and 7).

As described in §5, our work is a case study on CV but it may generalize to other DL application domains, such as Natural Language Processing and Audio. We observed no domain-specific steps in the proposed process, so we believe our case study should apply to other DL problem domains as well. The specific distribution of defects and problem-solving strategies may vary from domain to domain (*e.g.,* the available means of validation, such as the feasibility of visually inspecting results as done in CV), but we expect this DL reengineering process would not.

7.2 Comparison to Prior Works

We compare our work to three kinds of studies: those on DL model development, those on traditional software reengineering, and those on pre-trained model reuse. In Table 11 we summarize our analysis of the similarities and difficulties between our findings and these domains.

7.2.1 Deep Learning Model Development

Our general findings on the CV reengineering process match some results from prior works. For example, like prior bug studies on DL engineering (Islam et al. 2019; Humbatova et al. 2020), we observed a large percentage of API defects (21%, 74/348) within the CV reengineering process. In line with the results from Zhang *et al.* (Zhang et al. 2018), we also found basic defects are the most common phase.

Table 11 Differences between our findings on DL reengineering and prior work

Aspect	Prior findings	Our findings
Goals	Traditional software reengineering focuses on improving system quality, maintainability, and performance (Rosenberg and Hyatt 1996; Majthoub et al. 2018).	DL reengineering aims to facilitate further software reuse and support the ongoing evolution of the software. It is also part of the research-to-practice pipeline. (§4)
Model Implementation	Prior work highlighted the challenges in selecting model hyperparameters and debugging training process (Zhang et al. 2019, 2020b, 2018).	Performance debugging is also challging in DL reengineering process. Additional Challenges exist in model operationalization, portability of DL operations, and customized data pipeline.(§6.5)
Defect Distribution	Previous studies found API and basic defects common, with fewer hyperparameter and data quality defects (Islam et al. 2016; Humbatova et al. 2020).	DL reengineering has more defects related to hyper-parameter tuning and training data quality. (§6.1-§6.4)
Testing Practice	Pre-trained model reuse faces similar testing challenges but specific situations encountered are different (Braiek and Khomh 2020; Zhang et al. 2020a; Jiang et al. 2023b).	In DL reengineering, engineers can leverage existing implementations, so the application of effective testing methods, such as differential testing and metamorphic testing, is both feasible and can yield significant benefits. (§6.5, §7.1)

Within prior work, we consider DL development, traditional software reengineering, and pre-trained model reuse

However, we observed notably different proportions of defects, *e.g.,* by stage and by cause. We found a higher proportion of hyper-parameter tuning defects (5%, 18/348) in the DL reengineering process compared to the results of Islam *et al.*, who reported a proportion of less than 1% (Islam et al. 2019). Additionally, the results presented by Humbatova *et al.* shows that 95% of survey participants had trouble with training data (Humbatova et al. 2020). However, in our result, training data quality only accounts for 19% (13/68) of defects in data pipeline! This difference may arise from context: 58% (203/348) of the defects we analyzed were reported by re-users using benchmark datasets.

Qualitatively, our in-depth study of a CV reengineering team identified more detailed challenges in DL reengineering. For model configuration, we observed challenges in distinguishing models, identifying reported algorithm(s), and evaluating of trustworthiness. These were not mentioned in prior works (Islam et al. 2020). We also refined portability challenges into three different types: different names/signatures for the same behavior, inconsistent and undocumented behaviors, and different behavior on certain hardware. The latter two were not identified before (Zhang et al. 2019). Finally, we noted the importance — and difficulty — of performance debugging in DL reengineering.

Given these substantial differences, we provide a conjecture as to potential causes. DL reengineering process involves unique problem-solving strategies distinct from those used in normal DL development processes. These include the need to carefully review and modify hyperparameters from existing models and the data format to fit new datasets, which introduce additional complexity and the potential for data quality defects. Furthermore, identifying and rectifying defects even when using the same dataset can require advanced analytical skills

and understanding of the initial model's behaviors. The reusability and trustworthiness of the starting models also require careful assessment, requiring advanced knowledge of machine learning principles and critical evaluation of existing implementations. In contrast, the typical development process often focuses on designing and training a model from scratch, where control over variables and initial conditions is more straightforward (Amershi et al. 2019; Rahman et al. 2019). We conclude that problem-solving approaches in both the development and reengineering processes present distinct challenges: adapting existing components is a core engineering task; inventing new models leans more towards the realm of science.

7.2.2 Traditional Software Reengineering

We also compare the DL reengineering to other reengineering works in the software engineering literature (Singh et al. 2019; Bhavsar et al. 2020). We propose that the goal and focus are two main differences: First, the goal of many reengineering projects is to improve software functionality, performance, or implementation (Rosenberg and Hyatt 1996; Majthoub et al. 2018), while the main goal we saw in DL reengineering was to support further software reuse and customization. Second, other reengineering studies focus on the maintenance/process aspect, while in our study we saw that the DL reengineering activities focus on the implementation/evolution aspect (Bennett and Rajlich 2000). One possible causal factor here is that the DL reengineering activities we observed were building on research products, while general software reengineering is revisiting the output from engineering teams.

The research-to-practice pipeline has seen increasing adoption in the industry through the rapid invention and deployment of DL techniques (Wang et al. 2020b; Bubeck et al. 2023; Zhou et al. 2022; Dhanya et al. 2022). Our investigation reveals the role of DL reengineering process in the research-to-practice pipeline (Grima-Farrell 2017). As Davis *et al.* summarize, the reuse paradigm of DL models encompasses conceptual, adaptation, and deployment reuse (Davis et al. 2023). Our work draws attention to the significance of conceptual and adaptation reuse in the model reengineering process. In particular, our study underscores the ways in which practitioners transition knowledge from research prototypes into practical engineering implementations, such as replicating models in different frameworks or adapting models to custom datasets.

7.2.3 Pre-trained Deep Learning Model Reuse

Our insights on DL reengineering (§7.1) also highlight its similarities and differences compared to DL model reuse (§2.3). Similar to DL reengineering, pre-trained model reuse also encompasses several activities such as conceptual, adaptation, and deployment reuse (Davis et al. 2023). Both share common hurdles including lack of information about original prototypes, inconsistencies, and trust issues (Montes et al. 2022a; Jiang et al. 2023b). However, we note two difference between DL reengineering and pre-trained model reuse. First, DL reengineering can benefit significantly from effective differential and metamorphic testing techniques (§6.5.2), as compared to pre-trained model reuse. The goal of reengineering a model is to achieve the same performance when reusing or replicating models, while that of reusing pre-trained models is mainly focused on downstream performances (Jiang et al. 2023b). Second, prior work has explored effective methods for identifying suitable pre-trained models that can be adapted to specific downstream tasks (You et al. 2021b, a; Lu et al. 2019). The reengineering process also requires this model selection step (Fig. 12), and adaptation is a key component of the DL reengineering process. However, selecting a good

research prototype is harder in the reengineering process because model operationalization and portability of DL operations are the two most challenging parts. (§6.5.1).

Another emerging trend in reengineering models is the use of so-called "Foundation Models" (Zhou et al. 2023; Yuan et al. 2021). These models have unique attributes in model adaptation. For instance, large language models can be adapted using methods such as prompting or zero/few-shot learning (Touvron et al. 2023a, b; Brown et al. 2020; Liu et al. 2022; Wei et al. 2021; Abdullah et al. 2022). This adaptability distinguishes them from the more extended adaptation processes found in the DL reengineering techniques discussed in this work. The key advantage is that they leverage the knowledge and generalization capabilities embedded in their pretrained weights. This facilitates quicker and often more efficient adaptation to new tasks without the need for prolonged retraining or a customized data pipeline (Wang et al. 2023; Shu et al. 2022). We therefore expect that the focus of reengineering with foundation models will predominantly be on adaptation rather than reuse, replication, or enhancement. This assertion is based on the primary objective of employing foundation models for downstream tasks (Zhou et al. 2023). The portability of these foundation models is still an active area of research (Pan et al. 2023; Yuan 2023; Wu et al. 2023). Such unique characteristics might necessitate different reengineering practices tailored to foundation models. Regrettably, the reengineering phenomena of foundation models were not captured in our study because our data was collected in 2021, which was before the rise of foundation models such as LLMs.

7.3 Implications for Practice and Future Research

Our empirical data motivates many directions for further research. We divide these directions into suggestions for standardization (§7.3.1), opportunities for improved DL software testing (§7.3.2), further empirical study (§7.3.4), and techniques to enable model reuse (§7.3.3).

7.3.1 Standardized Practices

Direction 1: The state of practice in DL reengineering involves many challenges that can be tied to inadequate documentation and tooling. These challenges might be addressed with standardized documentation, checklists for replication, and automation that analyzes repositories and connects them to research papers.

As observed in the open-source defects, few of these reengineering efforts followed a standard process, causing novices to get confused. We recommend step-by-step guidelines for DL reengineering, starting with Fig. 12. Building on our findings, future studies could validate our proposed reengineering workflow, confirm the challenges, and evaluate the effectiveness of checklists and other process aids in improving DL reengineering.

Major companies have provided some practices to standardize the DL model documentation, such as model card proposed by Google (Mitchell et al. 2019) and metadata extraction from IBM (Tsay et al. 2022), our analysis (Fig. 10) and recent work (Jiang et al. 2023b) indicate that existing implementations still lack standardized documentation which results in the challenge of model analysis and reuse. Therefore, we recommend engineers develop and adhere to a *standard template*, *e.g.,* stating environment configuration and hyper-parameter tuning.

We also envision further studies on automated tools to extract the training scripts, hyperparameters, and evaluation results from open-source projects, aiming to enhance standardization

in the documentation. As an initial step in this direction, recent work developed an automated large-language model pipeline that can extract these metadata from model documentation (Jiang et al. 2024). However, their work shows the lack of metadata in the model cards. To address this, we recommend extending the metadata extraction tool from documentation to source code and to research papers, across which one might obtain more comprehensive metadata such as all configuration parameters. Another approach might be to develop a large language model with knowledge about DL reengineering and pre-trained models.

7.3.2 Support for Validation during DL Reengineering

Direction 2: The reengineering challenges we identified emphasize the need for better validation and debugging tools tailored to specific stages of DL reengineering. We observed particular gaps in the stages of Data Pipeline (*e.g.,* for data quality and augmentation defects) and Training (*e.g.,* for performance defects), as well as in the efficient development of unit and differential tests.

Our results shows the DL-stage-related characteristics of reengineering defects (§6.1–§6.4) and difficulties of debugging (§6.5). We recommend future directions on debugging tools for each DL stage, especially Data Pipeline and Training.

Our analysis shows that most of the defects in data pipeline are due to the incorrect data preprocessing and low data quality. There have been many studies on data management (Kumar et al. 2017), validation for DL datasets (Breck et al. 2019), and practical tools for data format converting (Willemink et al. 2020). However, we did not observe much use of these tools in (open-source) practice, including in the commercially maintained zoos. It is unclear whether the gap is in adopting existing tools or in meeting the real needs of practitioners. In addition, creating large DL datasets is expensive due to high labor and time cost. Data augmentation methods are widely used in recent DL models to solve the related problems (*e.g.,* overfitting, lower accuracy) (Shorten and Khoshgoftaar 2019). Our results (§6.4) show that engineers often encounter defects during data augmentation and data quality. Therefore, we recommend future studies on evaluating the quality of data augmentation. It would be of great help if engineers can test in advance whether the data augmentation is sufficient for training.

Researchers have been working on automated DL testing (Tian et al. 2018; Pei et al. 2017) and fuzzing (Zhang et al. 2021; Guo et al. 2018) tools for lowering the cost of the testing in the training stage. However, there are not many specific testing techniques for DL reengineering (Braiek and Khomh 2020). In our reengineering team (§6.5), we found performance debugging challenging, especially for reproducibility defects. We recommend software researchers explore end-to-end testing tools for reengineering tasks which can take advantage of existing model implementations. For example, end-to-end testing of reengineered models, particularly those replicated using a different DL framework, can be challenging. To address this, developing new fuzzing techniques offers promise (Jajal et al. 2023; Openja et al. 2022; Liu et al. 2023). By randomly generating inputs during the early training stage, these techniques can help predict the final performance of the replicated model compared to the original one. To lower end-to-end costs, improved unit and differential testing methods will also help. Our work highlights the need for further development in unit and differential testing techniques tailored specifically for deep learning models. Recent work has begun to study testing practices for such models (Ali et al. 2024). By focusing on different stages of the deep learning pipeline, new techniques can enable earlier detection and easier diagnosis of training bugs. For instance, differential testing can be employed to verify functionalities like model loading or loss function implementation by comparing the inputs and

outputs of the replicated model with the original one. Similarly, unit testing can be applied to visualize and validate individual components like the data pipeline or specific algorithms within the model, such as non-maximum suppression in object detection tasks (Gong et al. 2021).

7.3.3 Innovating Beyond Manual DL Model Reengineering

Direction 3: We found that practitioners perform DL model reengineering by hand, with limited automated support. Future work could examine automated model conversion from framework to framework, automated model component extraction for reuse, and domain-specific languages for DL mathematics.

DL Model reuse may mainly happen in the modeling stage. Most modeling defects are due to the operations of pre-trained weights and models (Fig. 10). Though there have been tools for the conversion of models between different DL frameworks, notably ONNX (ONN 2019a), converting models remains costly due to the rapidly increasing number of operators (Liu et al. 2020) and data types (ONN 2019b). Moreover, we found that engineers also struggle with data management and training configuration (§6.4, §6.5). Thus we recommend further investment in *end-to-end model conversion technology*.

Recent studies have shown a unique approach to reengineering DL models. Pan *et al.* propose a promising step to decompose DL models into reusable modules (Pan and Rajan 2020, 2022; Imtiaz et al. 2023). Imtiaz *et al.* proved that this decomposition approach can also support the reuse and replacement of recurrent neural networks (Imtiaz et al. 2023). These approaches show promise in mitigating the complexities and expenses typically associated with the DL reengineering process. We advocate for expanded research that leverages the distinctive characteristics of DL models to explore and develop innovative methodologies for facilitating DL reengineering.

When models cannot be reused automatically, DL reengineers must manually replicate the logic, resulting in a range of defects (Fig. 10). Both the open-source defects and team leaders mentioned mathematical defects, *e.g.,* sign mismatch or numerical instability (Fig. 9). Our findings indicated that mathematical functionalities are hard to verify (§6.4, §6.5) We suggest a domain-specific language for loss function mathematics, similar to how regular expressions support string parsing (Michael et al. 2019). While engineers currently leverage mathematical language to design loss functions, their implementation in general-purpose languages like Python appears to present verification difficulties. Our proposed DSL would offer a solution by facilitating the compilation of loss function code into a human-readable format, such as compiled LaTeX functions. Additionally, we recommend the development of automated test generation techniques specifically tailored for mathematical and numerical testing of these loss functions.

7.3.4 Empirical Studies

Direction 4: We found substantial disagreement between our results and prior work on DL failures, suggesting that our "process" focus may yield different results than a traditional "product"-focused failure analysis. Through the lens of DL reengineering, our study exposed many areas for deeper study, such as how individual classes of DL reengineering failures should be solved, and the compounding effect of discrepancies in upstream models. New datasets of DL models should simplify further study in this topic.

Given the substantial prior research on DL failures, both quantitative and qualitative, we were surprised by the differences between our results and prior work that we described in §7.2. Although we can only conjecture, one possible explanation of these differences is that prior work takes a product view in sampling data (cf. §2.1), while our work's sampling approach takes a process view guided by the engineering activity. The sampling methods between our work and previous studies vary in what they emphasize. Prior work used keywords to search for issues, pull requests, and Stack Overflow questions (Islam et al. 2019; Zhang et al. 2018; Humbatova et al. 2020; Sun et al. 2017; Shen et al. 2021) which provide them a product view of the deep learning failures. In contrast, during the data collection, we identified the engineering activities and process *first*, then categorized and analyzed the defects. If the results of a failure analysis differ based on whether data is sampled by a product or a process perspective, the implications for software failure studies are profound — most prior studies take a product view (Amusuo et al. 2022). This may bear further reflection by the empirical software engineering research community.

Our case study identified several gaps in our empirical understanding of DL reengineering. *First*, to fully understand reengineering problems, it would be useful to investigate more deeply how experts solve specific problems. Our data indicated the kinds of problems being solved during CV reengineering, but we did not explore the problem solution strategies nor the design rationale. For example, reengineering defects include indications of expert strategies for choosing loss functions[13] and tuning hyper-parameters[14]. *Second*, our findings motivate further interviews and surveys for DL engineers, *e.g.,* to understand the reengineering process workflow (Fig. 12) and to evaluate the efficiency benefits of our proposed strategies (among others (Serban et al. 2020)). *Third*, the difficulties we measured in DL model reengineering imply that reengineering is hard. As Montes et al. (2022b) showed, collections of pre-trained models (*e.g.,* ONNX model zoo (ONN 2019a) and Keras applications (Keras 2022)) may not be entirely consistent with the models they claim to replicate. Prior work also indicated that discrepancies exist in the DL model supply chain (Jiang et al. 2022b, 2023b). These defects could result in challenges of *model operationalization* and *portability* (§6.5.1) and therefore impact the downstream DL pipelines (Xin et al. 2021; Gopalakrishna et al. 2022; Nikitin et al. 2022). There is no comprehensive study on how these discrepancies can affect downstream implementations and what are effective ways to identify them.

In investigating these topics, future work can build on recent datasets such as PTMTorrent (Jiang et al. 2023c), PeaTMOSS (Jiang et al. 2024), and HFCommunity (Ait et al. 2023), which capture DL models, inter-model dependencies, and their downstream use on GitHub. These datasets can be used to analyze reengineering activities. For example, PeaTMOSS could be used to study how engineers adapt upstream pre-trained models to their downstream applications.

8 Threats to Validity

Construct Threats In this paper we introduced the concept of *Deep Learning Reengineering*. This concept tailors the idea of software reengineering (Linda et al. 1996; Byrne 1992) to the reengineering tasks pertinent to DL. We used this concept as a criterion by which to select repositories and defects for analysis. Although this concept resembles traditional problems in reuse and porting, we believe reengineering is a useful conceptualization for the activities of

[13] See https://github.com/ultralytics/yolov3/issues/2.

[14] See https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/issues/150.

software engineers working with DL technologies. Through our study, we provide empirical evidence on the prevalence and nature of these concepts, demonstrating the appropriateness of this conceptual framework for understanding and improving the practice of deep learning reengineering process (§4). We used saturation to assess the completeness of the taxonomies used to characterize DL reengineering defects.

We acknowledge that the definition and model of DL reengineering §4 used in our study (§3) may be perceived as overly broad. Ours is the first work to observe that reengineering is a major class of DL engineering work. Further work could examine the nuances between from-scratch development, direct reuse from papers or code via copy-paste, adaptation from one domain to another, adaptation between one framework and another, and other variations. This is a complex domain with many opportunities for study, *e.g.,* by varying study constructs and focuses.

In our study, we assert that repository forks are among the primary methods of DL reuse. However, recent research suggests that many engineers chose to reuse open-source pre-trained models and develop downstream models through fine-tuning or few-shot learning (Jiang et al. 2023b; Tan et al. 2018; Käding et al. 2017). The lack of direct evidence comparing the popularity of these two approaches introduces a construct validity threat, casting doubt on the representativeness of our repository mining approach. We recommend future studies to quantitatively measure and compare these methods of reuse.

Internal Threats In measuring aspects of this concept, we relied on manual classification of GitHub issue data. Our methodology might bring potential bias in our results. First, our data collection filters might bias the collection of defects. We followed previous studies to study closed issues, which allow us to understand the flow of a full discussion and can help to reveal all possible information types (Arya et al. 2019; Sun et al. 2017). However, we acknowledge that issues without associated fixes can also be important.

Second, the use of only one experienced rater in failure analysis and one analyzer in the interview study also increases the likelihood of subjective biases influencing the results. To mitigate subjectivity in the GitHub issue analysis, we adapted and applied existing taxonomies. Ambiguity was resolved through instrument calibration and discussion. Acceptable levels of interrater agreement were measured on a sample of the data.

We also agree that there could be some bias in the framework analysis of in the interview study. We considered the bias in our study design. To mitigate the bias in our interview data, we build a draft of reengineering workflow based on the knowledge of ML development workflow (Amershi et al. 2019). Our observations from the failure analysis also let us tease out similarities and differences in the reengineering context. During the interview, we asked if the subjects had anything to add to our workflow. We also conducted member checking by sharing our findings with 6 team leaders from §5.2.2, who confirmed their agreement with our analysis (Ritchie et al. 2013).

External Threats Our findings may not generalize to reengineering in other DL applications because our case study only considered DL reengineering in the context of computer vision. To mitigate this threat, we focused on the DL reengineering *process* and adapted general DL defect taxonomies from previous studies. The participants in our interview study have experience in multiple domains (Table 8) which can also mitigate this threat. Our findings indicate the distributions of reengineering phases (§6.1), defect types (§6.2), symptoms (§6.3), and root causes (§6.4) in the DL reengineering process (as measured on CV projects), as well as the challenges and practices we collected from the interview study (§6.5). We highlight two aspects of our work that may face particular generalizability threats: the taxonomies and distributions of defect symptoms (Fig. 9) and root causes (Fig. 10). Both of these may vary

by DL domain (*e.g.,* in what measurements are used to identify failure symptoms, and in what the failure modes of distinct architectures are).

Within our case study, our failure analysis examined open-source CV models. Our data may not generalize to commercial practices; to mitigate this, we focused on popular CV models. Lack of theoretical generalization is a drawback of our case study method; the trade-off is for a deeper look at a single context. We acknowledge that reengineering activities can vary significantly due to the diversity in training datasets, architectures, and learning methods (Tajbakhsh et al. 2020; Shrestha and Mahmood 2019; Khamparia and Singh 2019). As a result, necessary modifications to data pipelines, architectural adaptations, and replication checks may also differ. For instance, recent research on large language models has introduced low-rank adaptation methods specific to NLP contexts (Hu et al. 2021). However, while the specific methods or activities may differ, the overarching process aligns with the high-level framework outlined in this work (Fig. 1). This means that the specific defects we found in §6.1-§6.4 may vary, but our findings from a process view appear to be generalizable to DL reengineering in multiple DL application domains. As a point of comparison, we think that research on web applications contextualized to important frameworks, such as Rails (Yang et al. 2019) and Node.js (Chang et al. 2019), are worth having, even if not all software is a web application or if some web applications use other technologies.

Within our CV reengineering team study, our team had only two years of (undergraduate) experience in the domain. However, the corporate sponsor provided weekly expert coaching, and the results are now used internally by the sponsor. This provides some evidence that the team produced acceptable engineering results, implying that their process is worth describing in this work.

9 Conclusions

Software engineers are engaged in DL reengineering tasks — getting DL research ideas to work well in their own contexts. Prior work mainly focuses on the product view of DL systems, while our mixed-method case study explored the process view of characteristics, pitfalls, and practices of DL model reengineering activities. We quantitatively analyzed 348 CV reengineering defects and reported on a two-year reengineering effort. Our analysis shows that the characteristics of reengineering defects vary by DL stage, and that the environment and training configuration are the most challenging parts. Additionally, we conducted a qualitative analysis and identified four challenges of DL reengineering: model operationalization, performance debugging, portability of DL operations, and customizing data pipelines. Our findings from a process view appear to be generalizable to DL reengineering in multiple DL application domains, although further study is needed to confirm this point.

We integrated our findings through a DL reengineering process workflow, annotated with practices, challenges, and frequency of defects. We compared our work with prior studies on DL development, traditional software reengineering process, and pre-trained model reuse. Our work shares similar findings with prior defect studies from DL product perspectives, such as a large proportion of API defects. However, we found a higher proportion of defects caused by hyper-parameter tuning and training data quality. Moreover, we discovered and described similarities between model reengineering and pre-trained DL model reuse. There is some overlap between these two topics and the challenges and practices can also be shareable in both domains. Our results inform future research directions on the development of standardized

practices in the field, development of DL software testing tools, further exploration of model reengineering, and facilitating easier model reuse.

Data Availability Our artifact is at https://github.com/Wenxin-Jiang/EMSE-CVReengineering-Artifact.

Declarations

Disclosure of potential conflicts of interest This work was supported by Google and Cisco and by NSF awards #2107230, #2229703, #2107020, and #2104319. We acknowledge that Google has an interest in promoting the use of TensorFlow, hence their investment in the TensorFlow Model Garden. Although this interest enabled the collection of some of the data (the student team), Google did not otherwise engage in the project. No Google employees participated in the project, neither as subjects nor as researchers.

Research involving Human Participants Our human subjects work (interviews) followed a protocol that was approved by Purdue University's Institutional Review Board (IRB). The IRB protocol number is *#IRB-2021-366*.

References

ONNX (2019a) | Home. https://onnx.ai/

Portability between deep learning frameworks – with ONNX (2019b) https://blog.codecentric.de/en/2019/08/portability-deep-learning-frameworks-onnx/

Managing labels (2020) https://docs.github.com/en/issues/using-labels-and-milestones-to-track-work/managing-labels

Papers with Code - ML Reproducibility Challenge 2021 Edition (2020) https://paperswithcode.com/rc2021

Being a Computer Vision Engineer in 2021 (2021) https://viso.ai/computer-vision/computer-vision-engineer/

Machine Learning Operations (2021) https://ml-ops.org/

TensorFlow (2021) https://www.tensorflow.org/

Abdullah M, Madain A, Jararweh Y (2022) Chatgpt: Fundamentals, applications and social impacts. In: 2022 Ninth International conference on social networks analysis, management and security (SNAMS), IEEE, pp 1–8

Ait A, Izquierdo JLC, Cabot J (2023) Hfcommunity: A tool to analyze the hugging face hub community. In: 2023 IEEE International conference on software analysis, evolution and reengineering (SANER), IEEE, pp 728–732

Alahmari SS, Goldgof DB, Mouton PR, Hall LO (2020) Challenges for the Repeatability of Deep Learning Models. IEEE Access

AlDanial (2022) cloc. https://github.com/AlDanial/cloc

Ali Q, Riganelli O, Mariani L (2024) Testing in the evolving world of dl systems: Insights from python github projects. arXiv:2405.19976

Alzubaidi L, Zhang J, Humaidi AJ, Al-Dujaili A, Duan Y, Al-Shamma O, Santamaría J, Fadhel MA, Al-Amidie M, Farhan L (2021) Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions. J Big Data 8:1–74

Amershi S, Begel A, Bird C, DeLine R, Gall H (2019) Software Engineering for Machine Learning: A Case Study. In: International conference on software engineering: software engineering in practice (ICSE-SEIP)

Amusuo P, Sharma A, Rao SR, Vincent A, Davis JC (2022) Reflections on software failure analysis. In: ACM Joint european software engineering conference and symposium on the foundations of software engineering — Ideas, Visions, and Reflections track (ESEC/FSE-IVR)
Anandayuvaraj D, Davis JC (2022) Reflecting on recurring failures in iot development. In: Proceedings of the 37th IEEE/ACM International conference on automated software engineering, pp 1–5
Aranda J, Venolia G (2009) The secret life of bugs: Going past the errors and omissions in software repositories. In: International conference on software engineering (ICSE)
Arya D, Wang W, Guo JL, Cheng J (2019) Analysis and detection of information types of open source software issue discussions. In: 2019 IEEE/ACM 41st International conference on software engineering (ICSE), IEEE, pp 454–464
Bahdanau D, Cho KH, Bengio Y (2015) Neural machine translation by jointly learning to align and translate. In: International conference on learning representations (ICLR)
Banna V, Chinnakotla A, Yan Z, Vegesana A, Vivek N, Krishnappa K, Jiang W, Lu YH, Thiruvathukal GK, Davis JC (2021) An experience report on machine learning reproducibility: Guidance for practitioners and TensorFlow model garden contributors. https://arxiv.org/abs/2107.00821
Baysal O, Kononenko O, Holmes R, Godfrey MW (2012) The secret life of patches: A firefox case study. In: 2012 19th working conference on reverse engineering, IEEE, pp 447–455
Bennett KH, Rajlich VT (2000) Software maintenance and evolution: a roadmap. In: Proceedings of the conference on the future of software engineering, pp 73–87
Berner C, Brockman G, Chan B, Cheung V, Dębiak P, Dennison C, Farhi D, Fischer Q, Hashme S, Hesse C, Józefowicz R, Gray S, Olsson C, Pachocki J, Petrov M, Pinto HPdO, Raiman J, Salimans T, Schlatter J, Schneider J, Sidor S, Sutskever I, Tang J, Wolski F, Zhang S (2019) Dota 2 with Large Scale Deep Reinforcement Learning. arXiv:1912.06680
Bhatia A, Eghan EE, Grichi M, Cavanagh WG, Jiang ZM, Adams B (2023) Towards a change taxonomy for machine learning pipelines: Empirical study of ml pipelines and forks related to academic publications. Empirical Softw Eng 28(3):60
Bhavsar K, Shah V, Gopalan S (2020) Machine learning: a software process reengineering in software development organization. Int J Eng Advanced Technol 9(2):4492–4500
Bibal A, Frénay B (2016) Interpretability of Machine Learning Models and Representations: an Introduction. In: European symposium on artificial neural networks
Birt L, Scott S, Cavers D, Campbell C, Walter F (2016) Member checking: a tool to enhance trustworthiness or merely a nod to validation? Qualitative Health Res 26(13):1802–1811
Boehm B, Beck K (2010) The changing nature of software evolution; The inevitability of evolution. In: IEEE Software
Borges H, Valente MT (2018) What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform In: Journal of systems and software (JSS). https://doi.org/10.1016/j.jss.2018.09.016
Braiek HB, Khomh F (2020) On testing machine learning programs. J Syst Softw (JSS) 164:110542
Breck E, Cai S, Nielsen E, Salib M, Sculley D (2017) The ML test score: A rubric for ML production readiness and technical debt reduction. In: 2017 IEEE International conference on big data (big data), pp 1123–1132, https://doi.org/10.1109/BigData.2017.8258038
Breck E, Polyzotis N, Roy S, Whang S, Zinkevich M (2019) Data Validation for Machine Learning. In: the Conference on machine learning and systems (MLSys)
Brown TB, Mann B, Ryder N, Subbiah M, Kaplan J, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, Agarwal S, Herbert-Voss A, Krueger G, Henighan T, Child R, Ramesh A, Ziegler DM, Wu J, Winter C, Hesse C, Chen M, Sigler E, Litwin M, Gray S, Chess B, Clark J, Berner C, McCandlish S, Radford A, Sutskever I, Amodei D (2020) Language Models are Few-Shot Learners. Tech Rep arXiv:2005.14165
Bubeck S, Chandrasekaran V, Eldan R, Gehrke J, Horvitz E, Kamar E, Lee P, Lee YT, Li Y, Lundberg S et al (2023) Sparks of artificial general intelligence: Early experiments with gpt-4. arXiv:2303.12712
Byrne E (1992) A conceptual foundation for software re-engineering. In: Conference on software maintenance
Chang X, Dou W, Gao Y, Wang J, Wei J, Huang T (2019) Detecting atomicity violations for event-driven node.js applications. In: 2019 IEEE/ACM 41st International conference on software engineering (ICSE), IEEE, pp 631–642
Chen B, Wen M, Shi Y, Lin D, Rajbahadur GK, Ming Z, Jiang (2022a) Towards Training Reproducible Deep Learning Models. In: International conference on software engineering (ICSE), pp 2202–2214, https://doi.org/10.1145/3510003.3510163
Chen C, Liu MY, Tuzel O, Xiao J (2017) R-cnn for small object detection. In: Computer Vision–ACCV 2016: 13th Asian Conference on Computer Vision, Taipei, Taiwan, November 20-24, 2016, Revised Selected Papers, Part V 13, Springer, pp 214–230

Chen J, Liang Y, Shen Q, Jiang J (2022b) Toward Understanding Deep Learning Framework Bugs. arXiv:2203.04026
Chen K, Wang J, Pang J, Cao Y, Xiong Y, Li X, Sun S, Feng W, Liu Z, Xu J et al (2019) Mmdetection: Open mmlab detection toolbox and benchmark. arXiv:1906.07155
Cheng B, Misra I, Schwing AG, Kirillov A, Girdhar R (2022) Masked-attention Mask Transformer for Universal Image Segmentation. arXiv:2112.01527
Cohen D, Lindvall M, Costa P (2004) An introduction to agile methods. Advanced Comput 62(03):1–66
Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychological Measurement 20(1):37–46
Davis JC, Jajal P, Jiang W, Schorlemmer TR, Synovic N, Thiruvathukal GK (2023) Reusing deep learning models: Challenges and directions in software engineering. In: Proceedings of the IEEE john vincent atanasoff symposium on modern computing (JVA'23)
Devanbu P, Dwyer M, Elbaum S, Lowry M, Moran K, Poshyvanyk D, Ray B, Singh R, Zhang X (2020) Deep Learning & Software Engineering: State of Research and Future Directions. arXiv:2009.08525
Dhanya V, Subeesh A, Kushwaha N, Vishwakarma DK, Kumar TN, Ritika G, Singh A (2022) Deep learning based computer vision approaches for smart agricultural applications. Artif Intell Agric
Ding Z, Reddy A, Joshi A (2021) Reproducibility. https://blog.ml.cmu.edu/2020/08/31/5-reproducibility/
Doshi-Velez F, Kim B (2017) Towards A Rigorous Science of Interpretable Machine Learning. arXiv:1702.08608
Eghbali A, Pradel M (2020) No strings attached: an empirical study of string-related software bugs. In: International conference on automated software engineering (ASE)
Face H (2024) Hugging Face Documentation: timm. https://huggingface.co/docs/timm/index
Fitzgerald B (2006) The transformation of open source software. MIS Quarterly pp 587–598
Forsyth DA, Ponce J (2002) Computer vision: a modern approach. Prentice Hall Professional Technical Reference
Garcia J, Feng Y, Shen J, Almanee S, Xia Y, Chen QA (2020) A comprehensive study of autonomous vehicle bugs. In: International Conference on Software Engineering (ICSE), https://dl.acm.org/doi/10.1145/3377811.3380397
Gharehyazie M, Ray B, Filkov V (2017) Some from here, some from there: Cross-project code reuse in github. In: 2017 IEEE/ACM 14th International conference on mining software repositories (MSR), IEEE, pp 291–301
Goel A, Tung C, Lu YH, Thiruvathukal GK (2020) A Survey of Methods for Low-Power Deep Learning and Computer Vision. In: IEEE World forum on internet of things (WF-IoT)
Gong M, Wang D, Zhao X, Guo H, Luo D, Song M (2021) A review of non-maximum suppression algorithms for deep learning target detection. Seventh symposium on novel photoelectronic detection technology and applications, SPIE 11763:821–828
Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT press
Google (2022) Tensorflow model garden. https://github.com/tensorflow/models
Gopalakrishna NK, Anandayuvaraj D, Detti A, Bland FL, Rahaman S, Davis JC (2022) "If security is required": engineering and security practices for machine learning-based IoT devices. In: Proceedings of the 4th international workshop on software engineering research & practices for the internet of things (SERP4IoT), pp 1–8
Goyal P, Dollár P, Girshick R, Noordhuis P, Wesolowski L, Kyrola A, Tulloch A, Jia Y, He K (2018) Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. arXiv:1706.02677
Grima-Farrell C (2017) The rtp model: An interactive research to practice framework. What Matters in a Research to Practice Cycle? Teachers as Researchers pp 237–250
Guan H, Xiao Y, Li J, Liu Y, Bai G (2023) A comprehensive study of real-world bugs in machine learning model optimization. In: Proceedings of the international conference on software engineering
Gundersen OE, Kjensmo S (2018) State of the art: Reproducibility in artificial intelligence. AAAI Conference on Artif Intell (AAAI)
Gundersen OE, Gil Y, Aha DW (2018) On reproducible AI: Towards reproducible research, open science, and digital scholarship in AI publications. AI Magazine
Guo J, Jiang Y, Zhao Y, Chen Q, Sun J (2018) DLFuzz: Differential Fuzzing Testing of Deep Learning Systems. In: European Software engineering conference and symposium on the foundations of software engineering (ESEC/FSE)
Hu EJ, Shen Y, Wallis P, Allen-Zhu Z, Li Y, Wang S, Wang L, Chen W (2021) Lora: Low-rank adaptation of large language models. arXiv:2106.09685
Humbatova N, Jahangirova G, Bavota G, Riccio V, Stocco A, Tonella P (2020) Taxonomy of real faults in deep learning systems. In: International conference on software engineering (ICSE)
Hutson M (2018) Artificial intelligence faces reproducibility crisis. American Assoc Advancement Sci 359(6377):725–726. https://doi.org/10.1126/science.359.6377.725

Imtiaz SM, Batole F, Singh A, Pan R, Cruz BD, Rajan H (2023) Decomposing a recurrent neural network into modules for enabling reusability and replacement. In: 2023 IEEE/ACM 45th International conference on software engineering (ICSE), IEEE, pp 1020–1032
Islam JF, Mondal M, Roy CK (2016) Bug replication in code clones: An empirical study. In: International conference on software analysis, evolution, and reengineering (SANER), IEEE, vol 1, pp 68–78
Islam MJ, Nguyen G, Pan R, Rajan H (2019) A comprehensive study on deep learning bug characteristics. In: European software engineering conference and symposium on the foundations of software engineering (ESEC/FSE)
Islam MJ, Pan R, Nguyen G, Rajan H (2020) Repairing deep neural networks: fix patterns and challenges. In: International conference on software engineering (ICSE)
Jajal P, Jiang W, Tewari A, Woo J, Lu YH, Thiruvathukal GK, Davis JC (2023) Analysis of failures and risks in deep learning model converters: A case study in the onnx ecosystem. arXiv:2303.17708
Jarzabek S (1993) Software reengineering for reusability. In: International computer software and applications conference (COMPSAC)
Jiang W, Synovic N, Sethi R (2022a) An Empirical Study of Artifacts and Security Risks in the Pre-trained Model Supply Chain. Los Angeles p 10
Jiang W, Synovic N, Sethi R, Indarapu A, Hyatt M, Schorlemmer TR, Thiruvathukal GK, Davis JC (2022b) An empirical study of artifacts and security risks in the pre-trained model supply chain. In: ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED'22), p 105–114, https://doi.org/10.1145/3560835.3564547
Jiang W, Cheung C, Kim M, Kim H, Thiruvathukal GK, Davis JC (2023a) Naming practices of pre-trained models in hugging face. arXiv:2310.01642
Jiang W, Synovic N, Hyatt M, Schorlemmer TR, Sethi R, Lu YH, Thiruvathukal GK, Davis JC (2023b) An empirical study of pre-trained model reuse in the hugging face deep learning model registry. In: IEEE/ACM 45th International conference on software engineering (ICSE'23)
Jiang W, Synovic N, Jajal P, Schorlemmer TR, Tewari A, Pareek B, Thiruvathukal GK, Davis JC (2023c) Ptmtorrent: A dataset for mining open-source pre-trained model packages. Proceedings of the 20th International Conference on Mining Software Repositories (MSR'23)
Jiang W, Yasmin J, Jones J, Synovic N, Kuo J, Bielanski N, Tian Y, , Thiruvathukal GK, Davis JC (2024) Peatmoss: A dataset and initial analysis of pre-trained models in open-source software. In: International conference on mining software repositories (MSR)
Jing YK (2021) Model Zoo - Deep learning code and pretrained models. https://modelzoo.co/
Johnson RB, Onwuegbuzie AJ (2004) Mixed methods research: a research paradigm whose time has come. Educ Res 33(7):14–26
Käding C, Rodner E, Freytag A, Denzler J (2017) Fine-Tuning Deep Neural Networks in Continuous Learning Scenarios. In: Chen CS, Lu J, Ma KK (eds) Computer Vision – ACCV 2016 Workshops, vol 10118, Springer International Publishing, Cham, pp 588–605, https://doi.org/10.1007/978-3-319-54526-4_43, http://link.springer.com/10.1007/978-3-319-54526-4_43, series Title: Lecture Notes in Computer Science
Keras (2022) Keras applications. https://keras.io/api/applications/
Keshta N, Morgan Y (2017) Comparison between traditional plan-based and agile software processes according to team size & project domain (a systematic literature review). In: 2017 8th IEEE Annual information technology, electronics and mobile communication conference (IEMCON), IEEE, pp 567–575
Khamparia A, Singh KM (2019) A systematic review on deep learning architectures and applications. Expert Syst 36(3):e12400
Kim J, Li J (2020) Introducing the model garden for tensorflow 2. https://blog.tensorflow.org/2020/03/introducing-model-garden-for-tensorflow-2.html
Krizhevsky A, Sutskever I, Hinton GE (2012) ImageNet Classification with Deep Convolutional Neural Networks. Advances in neural information processing systems (NeurIPS) 6:84–90
Kumar A, Boehm M, Yang J (2017) Data Management in Machine Learning: Challenges, Techniques, and Systems. In: International conference on management of data
Kuutti S, Bowden R, Jin Y, Barber P, Fallah S (2020) A survey of deep learning applications to autonomous vehicle control. IEEE Trans Intell Transportation Syst 22(2):712–733
Leveson NG (1995) Safeware: System safety and computers. ACM, New York, NY, USA
Leveson NG (2016) Engineering a safer world: Systems thinking applied to safety. The MIT Press
Li R, Jiao Q, Cao W, Wong HS, Wu S (2020) Model Adaptation: Unsupervised Domain Adaptation without Source Data. Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition pp 9638–9647,https://doi.org/10.1109/CVPR42600.2020.00966
Li Z, Liu F, Yang W, Peng S, Zhou J (2021) A survey of convolutional neural networks: analysis, applications, and prospects. IEEE Trans Neural Netw Learn Syst

Lin TY, Maire M, Belongie S et al (2014) Microsoft COCO: Common Objects in Context. In: European conference on computer vision (ECCV)
Linda D, Rosenberg H, Hyatt LE (1996) Software Re-engineering. Softw Assurance Technol Center
Liu C, Gao C, Xia X, Lo D, Grundy J, Yang X (2021) On the Replicability and Reproducibility of Deep Learning in Software Engineering. ACM Trans Softw Eng Methodol 31(1):1–46
Liu J, Lin J, Ruffy F, Tan C, Li J, Panda A, Zhang L (2023) Nnsmith: Generating diverse and valid test cases for deep learning compilers. In: Proceedings of the 28th ACM international conference on architectural support for programming languages and operating systems, Volume 2, pp 530–543
Liu X, Ji K, Fu Y, Tam W, Du Z, Yang Z, Tang J (2022) P-tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks. In: Proceedings of the 60th annual meeting of the association for computational linguistics (Volume 2: Short Papers), pp 61–68
Liu Y, Xu C, Cheung SC (2014) Characterizing and detecting performance bugs for smartphone applications. In: Proceedings of the 36th International Conference on Software Engineering, ACM, Hyderabad India, pp 1013–1024, https://doi.org/10.1145/2568225.2568229, https://dl.acm.org/doi/10.1145/2568225.2568229
Liu Y, Chen C, Zhang R, Qin T, Ji X, Lin H, Yang M (2020) Enhancing the interoperability between deep learning frameworks by model conversion. In: European software engineering conference/foundations of software engineering (ESEC/FSE)
Lorenzoni G, Alencar P, Nascimento N, Cowan D (2021) Machine Learning Model Development from a Software Engineering Perspective: A Systematic Literature Review. arXiv:2102.07574
Lu B, Yang J, Chen LY, Ren S (2019) Automating Deep Neural Network Model Selection for Edge Inference. In: 2019 IEEE First International conference on cognitive machine intelligence (CogMI), pp 184–193, https://doi.org/10.1109/CogMI48466.2019.00035
Ma L, Juefei-Xu F, Zhang F, Sun J, Xue M, Li B, Chen C, Su T, Li L, Liu Y et al (2018a) Deepgauge: Multi-granularity testing criteria for deep learning systems. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pp 120–131
Ma S, Liu Y, Lee WC, Zhang X, Grama A (2018b) Mode: automated neural network model debugging via state differential analysis and input selection. In: Proceedings of the 2018 26th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 175–186
Majthoub M, Qutqut MH, Odeh Y (2018) Software re-engineering: An overview. In: 2018 8th International conference on computer science and information technology (CSIT), IEEE, pp 266–270
McHugh ML (2012) Interrater reliability: the kappa statistic. Biochemia Medica 22(3):276–282
Mckeeman WM (1998) Differential Testing for Software. Digital Technical J
Meta (2022) Torchvision. https://github.com/pytorch/vision
Meta (2024a) Detectron. https://ai.meta.com/tools/detectron/
Meta (2024b) Detectron2. https://ai.meta.com/tools/detectron2/
Michael LG, Donohue J, Davis JC, Lee D, Servant F (2019) Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 415–426, https://doi.org/10.1109/ASE.2019.00047, iSSN: 2643-1572
Mitchell M, Wu S, Zaldivar A, Barnes P, Vasserman L, Hutchinson B, Spitzer E, Raji ID, Gebru T (2019) Model Cards for Model Reporting. In: Proceedings of the Conference on Fairness, Accountability, and Transparency, ACM, Atlanta GA USA, pp 220–229, https://doi.org/10.1145/3287560.3287596, https://dl.acm.org/doi/10.1145/3287560.3287596
Montes D, Peerapatanapokin P, Schultz J, Guo C, Jiang W, Davis JC (2022a) Discrepancies among pre-trained deep neural networks: a new threat to model zoo reliability. In: European software engineering conference and symposium on the foundations of software engineering (ESEC/FSE-IVR track)
Montes D, Pongpatapee P, Schultz J, Guo C, Jiang W, Davis J (2022b) Discrepancies among Pre-trained Deep Neural Networks: A New Threat to Model Zoo Reliability. In: ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering — Ideas, Visions, and Reflections track (ESEC/FSE-IVR)
Nahar N, Zhou S, Lewis G, Kästner C (2022) Collaboration Challenges in Building ML-Enabled Systems: Communication, Documentation, Engineering, and Process. In: International conference on software engineering (ICSE)
Nepal U, Eslamiat H (2022) Comparing yolov3, yolov4 and yolov5 for autonomous landing spot detection in faulty uavs. Sensors 22(2):464
Nikanjam A, Khomh F (2021) Design Smells in Deep Learning Programs: An Empirical Study. In: IEEE International conference on software maintenance and evolution (ICSME)

Nikitin NO, Vychuzhanin P, Sarafanov M, Polonskaia IS, Revin I, Barabanova IV, Maximov G, Kalyuzhnaya AV, Boukhanovsky A (2022) Automated evolutionary approach for the design of composite machine learning pipelines. Future Generation Computer Systems

O'Connor R (2023) Pytorch vs tensorflow in 2023. https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023/

Openja M, Nikanjam A, Yahmed AH, Khomh F, Jiang ZMJ (2022) An Empirical Study of Challenges in Converting Deep Learning Models. In: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 13–23, https://doi.org/10.1109/ICSME55016.2022.00010, iSSN: 2576-3148

O'Shea K, Nash R (2015) An introduction to convolutional neural networks. arXiv:1511.08458

Pan R, Rajan H (2020) On decomposing a deep neural network into modules. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 889–900

Pan R, Rajan H (2022) Decomposing convolutional neural networks into reusable and replaceable modules. In: International conference on software engineering (ICSE), ACM, Pittsburgh Pennsylvania

Pan R, Ibrahimzada AR, Krishna R, Sankar D, Wassi LP, Merler M, Sobolev B, Pavuluri R, Sinha S, Jabbarvand R (2023) Understanding the effectiveness of large language models in code translation. arXiv:2308.03109

Panchal D, Baran I, Musgrove D, Lu D (2023) Mlops: Automatic, zero-touch and reusable machine learning training and serving pipelines. In: 2023 IEEE International conference on internet of things and intelligence systems (IoTaIS), IEEE, pp 175–181

Panchal D, Verma P, Baran I, Musgrove D, Lu D (2024) Reusable mlops: Reusable deployment, reusable infrastructure and hot-swappable machine learning models and services. arXiv:2403.00787

Pei K, Cao Y, Yang J, Jana S (2017) DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In: Symposium on operating systems principles (SOSP)

Perry D, Sim S, Easterbrook S (2004) Case studies for software engineers. In: International conference on software engineering (ICSE)

Pham HV, Qian S, Wang J, Lutellier T, Rosenthal J, Tan L, Yu Y, Nagappan N (2020) Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance. In: International conference on automated software engineering (ASE), https://doi.org/10.1145/3324884.3416545

Pineau J (2022) How the AI community can get serious about reproducibility. https://ai.facebook.com/blog/how-the-ai-community-can-get-serious-about-reproducibility/

Pineau J, Vincent-Lamarre P, Sinha K, Lariviere V, Beygelzimer A (2020) Improving Reproducibility in Machine Learning Research. J Machine Learn Res

Popel M, Tomkova M, Tomek J, Kaiser Ł, Uszkoreit J, Bojar O, Žabokrtskỳ Z (2020) Transforming machine translation: a deep learning system reaches news translation quality comparable to human professionals. Nature Commun 11(1):1–15

Pressman RS (2005) Software engineering: a practitioner's approach. Palgrave Macmillan

Pytorch (2021) Pytorch hub. https://pytorch.org/hub/

Qi B, Sun H, Gao X, Zhang H, Li Z, Liu X (2023) Reusing deep neural network models through model re-engineering. In: International Conference on Software Engineering, IEEE Press, p 983–994, https://doi.org/10.1109/ICSE48619.2023.00090

Rahman S, River E, Khomh F, Guhneuc YG, Lehnert B (2019) Machine learning software engineering in practice: An industrial case study. arXiv preprint https://doi.org/10.48550/arXiv.1906.07154

Ralph P, Ali Nb, Baltes S, Bianculli D, Diaz J, Dittrich Y, Ernst N, Felderer M, Feldt R, Filieri A, de França BBN, Furia CA, Gay G, Gold N, Graziotin D, He P, Hoda R, Juristo N, Kitchenham B, Lenarduzzi V, Martínez J, Melegati J, Mendez D, Menzies T, Molleri J, Pfahl D, Robbes R, Russo D, Saarimäki N, Sarro F, Taibi D, Siegmund J, Spinellis D, Staron M, Stol K, Storey MA, Taibi D, Tamburri D, Torchiano M, Treude C, Turhan B, Wang X, Vegas S (2021) Empirical Standards for Software Engineering Research. arXiv:2010.03525

Redmon J, Divvala S, Girshick R, Farhadi A (2016) You only look once: unified, real-time object detection. In: Proceedings of the IEEE conference on computer vision and pattern recognition, 779–788

Ren S, He K, Girshick R, Sun J (2017) Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. IEEE Trans Pattern Anal Machine Intell (TPAMI)

Ritchie J, Spencer L (2002) Qualitative data analysis for applied policy research. In: Analyzing qualitative data, Routledge, pp 187–208

Ritchie J, Lewis J, Nicholls CM, Ormston R et al (2013) Qualitative research practice: A guide for social science students and researchers. Sage

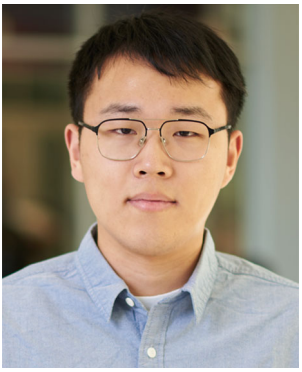
Rosenberg LH, Hyatt LE (1996) Software re-engineering. Software Assurance Technology Center pp 2–3

Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empirical Softw Eng (EMSE)

Saha RK, Khurshid S, Perry DE (2014) An empirical study of long lived bugs. 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014 - Proceedings pp 144–153, https://doi.org/10.1109/CSMR-WCRE.2014.6747164
Schelter S, Boese JH, Kirschnick J, Klein T, Seufert S (2017) Automatically tracking metadata and provenance of machine learning experiments. In: Machine learning systems workshop at NIPS
Schelter S, Biessmann F, Januschowski T, Salinas D, Seufert S, Szarvas G (2018) On Challenges in Machine Learning Model Management. Bullet IEEE Computer Soc Technical Committee Data Eng
Schmidhuber J (2015) Deep learning in neural networks: An overview. Neural Netw
Sculley D, Holt G, Golovin D, Davydov E, Phillips T, Ebner D, Chaudhary V, Young M (2014) Machine Learning : The High-Interest Credit Card of Technical Debt. In: NIPS Workshop on software engineering for machine learning (SE4ML)
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, Curran Associates, Inc., vol 28, https://proceedings.neurips.cc/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html
Seaman CB, Shull F, Regardie M, Elbert D, Feldmann RL, Guo Y, Godfrey S (2008) Defect categorization: making use of a decade of widely varying historical data. In: Empirical software engineering and measurement (ESEM)
Serban A, Van Der Blom K, Hoos H, Visser J (2020) Adoption and effects of software engineering best practices in machine learning. Int Symposium on Empirical Softw Eng Measurement 10(1145/3382494):3410681
Shen Q, Ma H, Chen J, Tian Y, Cheung SC, Chen X (2021) A comprehensive study of deep learning compiler bugs. In: European software engineering conference and symposium on the foundations of software engineering (ESEC/FSE)
Shorten C, Khoshgoftaar TM (2019) A survey on Image Data Augmentation for Deep Learning. J Big Data
Shrestha A, Mahmood A (2019) Review of deep learning algorithms and architectures. IEEE access 7:53040–53065
Shu M, Nie W, Huang DA, Yu Z, Goldstein T, Anandkumar A, Xiao C (2022) Test-time prompt tuning for zero-shot generalization in vision-language models. Adv Neural Inf Process Syst 35:14274–14289
Singh J, Singh K, Singh J (2019) Reengineering framework for open source software using decision tree approach. Int J Electrical Computer Eng (IJECE) 9(3):2041–2048
Srivastava A, Thomson S (2009) Framework analysis: A qualitative methodology for applied policy research
Sun X, Zhou T, Li G, Hu J, Yang H, Li B (2017) An Empirical Study on Real Bugs for Machine Learning Programs. In: Asia-Pacific SOFTWARE ENGINEERING CONFERENCE (APSEC)
Szeliski R (2022) Computer vision: algorithms and applications. Springer Nature
Taecharungroj V (2023) "what can chatgpt do?" analyzing early reactions to the innovative ai chatbot on twitter. Big Data Cognitive Comput 7(1):35
Tajbakhsh N, Jeyaseelan L, Li Q, Chiang JN, Wu Z, Ding X (2020) Embracing imperfect datasets: A review of deep learning solutions for medical image segmentation. Med Image Anal 63:101693
Tan C, Sun F, Kong T, Zhang W, Yang C, Liu C (2018) A Survey on Deep Transfer Learning. IEEE Trans Knowl Data Eng
Tan L, Liu C, Li Z, Wang X, Zhou Y, Zhai C (2014) Bug characteristics in open source software. Empirical Softw Eng (EMSE)
Tatman R, Vanderplas J, Dane S (2018) A Practical Taxonomy of Reproducibility for Machine Learning Research. In: Reproducibility in machine learning workshop at ICML
Thiruvathukal GK, Lu YH, Kim J, Chen Y, Chen B (2022) Low-power Computer Vision: Improve the Efficiency of Artificial Intelligence
Thung F, Wang S, Lo D, Jiang L (2012) An empirical study of bugs in machine learning systems. In: International symposium on software reliability engineering (ISSRE)
Tian Y, Pei K, Jana S, Ray B (2018) DeepTest: automated testing of deep-neural-network-driven autonomous cars. In: International conference on software engineering (ICSE)
Touvron H, Lavril T, Izacard G, Martinet X, Lachaux MA, Lacroix T, Rozière B, Goyal N, Hambro E, Azhar F et al (2023a) Llama: Open and efficient foundation language models. arXiv:2302.13971
Touvron H, Martin L, Stone K, Albert P, Almahairi A, Babaei Y, Bashlykov N, Batra S, Bhargava P, Bhosale S et al (2023b) Llama 2: Open foundation and fine-tuned chat models. arXiv:2307.09288
Tsay J, Braz A, Hirzel M, Shinnar A, Mummert T (2020) AIMMX: Artificial Intelligence Model Metadata Extractor. In: International conference on mining software repositories (MSR), https://doi.org/10.1145/3379597.3387448
Tsay J, Braz A, Hirzel M, Shinnar A, Mummert T (2022) Extracting enhanced artificial intelligence model metadata from software repositories. Empirical Softw Eng 27(7):176. https://doi.org/10.1007/s10664-022-10206-6, https://link.springer.com/10.1007/s10664-022-10206-6

Tucker DC, Devon MS (2010) A Case Study in Software Reengineering. In: International conference on informatio (itng)n technology: New Generations
Unceta I, Nin J, Pujol O (2020) Environmental adaptation and differential replication in machine learning. Entropy 22(10):1122
Valett JD, McGarry FE (1989) A Summary of Software Measurement Experiences in the Software Engineering Laboratory. J Syst Softw 9:137–148
Vartak M, Subramanyam H, Lee WE, Viswanathan S, Husnoo S, Madden S, Zaharia M (2016) Modeldb: a system for machine learning model management. In: the workshop on human-in-the-loop data analytics
Villa J, Zimmerman Y (2018) Reproducibility in ML: why it matters and how to achieve it. https://determined.ai/blog/reproducibility-in-ml
Vinyals O, Babuschkin I, Czarnecki WM, Mathieu M, Dudzik A, Chung J, Choi DH, Powell R, Ewalds T, Georgiev P et al (2019) Grandmaster level in starcraft ii using multi-agent reinforcement learning. Nature
Voulodimos A, Doulamis N, Doulamis A, Protopapadakis E (2018) Deep Learning for Computer Vision: A Brief Review. Comput Intell Neurosci
Wang J, Dou W, Gao Y, Gao C, Qin F, Yin K, Wei J (2017) A comprehensive study on real world concurrency bugs in Node.js. In: 2017 32nd IEEE/ACM International conference on automated software engineering (ASE), pp 520–531, https://doi.org/10.1109/ASE.2017.8115663
Wang J, Ma Y, Zhang L, Gao RX, Wu D (2018) Deep learning for smart manufacturing: Methods and applications. J Manufac Syst 48:144–156
Wang J, Lu Y, Yuan B, Chen B, Liang P, De Sa C, Re C, Zhang C (2023) Cocktailsgd: Fine-tuning foundation models over 500mbps networks. In: International conference on machine learning, PMLR, pp 36058–36076
Wang P, Brown C, Jennings JA, Stolee KT (2020a) An Empirical Study on Regular Expression Bugs. In: International conference on mining software repositories (MSR)
Wang S, Huang L, Ge J, Zhang T, Feng H, Li M, Zhang H, Ng V (2020b) Synergy between machine/deep learning and software engineering: How far are we? arXiv:2008.05515
Wardat M, Le W, Rajan H (2021) DeepLocalize: Fault Localization for Deep Neural Networks. In: 2021 IEEE/ACM 43rd International conference on software engineering (ICSE), pp 251–262, https://doi.org/10.1109/ICSE43902.2021.00034
Wei J, Bosma M, Zhao VY, Guu K, Yu AW, Lester B, Du N, Dai AM, Le QV (2021) Finetuned language models are zero-shot learners. arXiv:2109.01652
Wei Z, Wang H, Yang Z, Chan W (2022) Sebox4dl: a modular software engineering toolbox for deep learning models. In: Proceedings of the ACM/IEEE 44th International conference on software engineering: companion proceedings, pp 193–196
Willemink MJ, Koszek WA, Hardell C, Wu J, Fleischmann D, Harvey H, Folio LR, Summers RM, Rubin DL, Lungren MP (2020) Preparing Medical Imaging Data for Machine Learning. Radiological Society of North America
Wolf T, Debut L, Sanh V, Chaumond J, Delangue C, Moi A, Cistac P, Rault T, Louf R, Funtowicz M, Davison J, Shleifer S, von Platen P, Ma C, Jernite Y, Plu J, Xu C, Le Scao T, Gugger S, Drame M, Lhoest Q, Rush A (2020) Transformers: State-of-the-Art Natural Language Processing. In: Conference on empirical methods in natural language processing: system demonstrations
Wu C, Yin S, Qi W, Wang X, Tang Z, Duan N (2023) Visual chatgpt: Talking, drawing and editing with visual foundation models. arXiv:2303.04671
Wu Y, Schuster M, Chen Z, Le QV, Norouzi M, Macherey W, Krikun M, Cao Y, Gao Q, Macherey K et al (2016) Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv
Xin D, Miao H, Parameswaran A, Polyzotis N (2021) Production machine learning pipelines: Empirical analysis and optimization opportunities. In: Proceedings of the 2021 international conference on management of data, pp 2639–2652
Xu S, Wang J, Shou W, Ngo T, Sadick AM, Wang X (2021) Computer Vision Techniques in Construction: A Critical Review. Archives of Computational Methods in Engineering
Yang J, Yan C, Wan C, Lu S, Cheung A (2019) View-centric performance optimization for database-backed web applications. In: 2019 IEEE/ACM 41st International conference on software engineering (ICSE), IEEE, pp 994–1004
You K, Liu Y, Wang J, Jordan MI, Long M (2021a) Ranking and Tuning Pre-trained Models: A New Paradigm of Exploiting Model Hubs. J Machine Learn Res (JMLR) 23(1):9400–9446, arXiv:2110.10545
You K, Liu Y, Wang J, Long M (2021b) LogME: Practical Assessment of Pre-trained Models for Transfer Learning. In: International conference on machine learning (ICML), PMLR, pp 12133–12143, https://proceedings.mlr.press/v139/you21b.html

Yuan L, Chen D, Chen YL, Codella N, Dai X, Gao J, Hu H, Huang X, Li B, Li C et al (2021) Florence: A new foundation model for computer vision. arXiv:2111.11432
Yuan Y (2023) On the power of foundation models. In: International conference on machine learning, PMLR, pp 40519–40530
Zhang JM, Harman M, Ma L, Liu Y (2020) Machine learning testing: Survey, landscapes and horizons. IEEE Trans Softw Eng 48(1):1–36
Zhang R, Xiao W, Zhang H, Liu Y, Lin H, Yang M (2020b) An empirical study on program failures of deep learning jobs. In: International conference on software engineering (ICSE)
Zhang T, Gao C, Ma L, Lyu M, Kim M (2019) An Empirical Study of Common Challenges in Developing Deep Learning Applications. In: International symposium on software reliability engineering (ISSRE)
Zhang X, Liu J, Sun N, Fang C, Liu J, Wang J, Chai D, Chen Z (2021) Duo: Differential Fuzzing for Deep Learning Operators. IEEE Trans Reliability
Zhang Y, Chen Y, Cheung SC, Xiong Y, Zhang L (2018) An empirical study on TensorFlow program bugs. Int Symposium Soft Testing Anal (ISSTA)
Zhang Y, Ren L, Chen L, Xiong Y, Cheung SC, Xie T (2020c) Detecting numerical bugs in neural network architectures. European software engineering conference and symposium on the foundations of software engineering (ESEC/FSE)
Zhou C, Li Q, Li C, Yu J, Liu Y, Wang G, Zhang K, Ji C, Yan Q, He L et al (2023) A comprehensive survey on pretrained foundation models: A history from bert to chatgpt. arXiv:2302.09419
Zhou L, Zhang L, Konz N (2022) Computer vision techniques in manufacturing. IEEE Trans Syst, Man, and Cybernetics: Syst 53(1):105–117
Zou X, Yang J, Zhang H, Li F, Li L, Gao J, Lee YJ (2023) Segment Everything Everywhere All at Once. arXiv:2304.06718

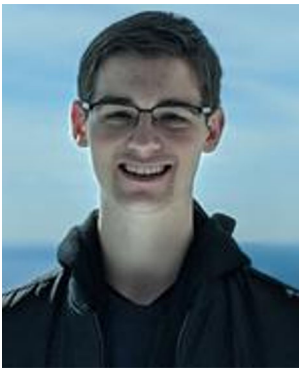


Wenxin Jiang is a Ph.D. candidate in the Elmore Family School of Electrical & Computer Engineering at Purdue University, under the supervision of Prof. James Davis. His research primarily focuses on Software Engineering for AI, with an emphasis on open-source pre-trained AI models. He is also interested in machine learning systems, software supply chain security, and trustworthy/responsible AI. Contact him at jiang784@purdue.edu.

Vishnu Banna got a BSc in Computer Engineering from Purdue University. He is currently a Deep Learning Engineer at Apple. He is interested in Deep Learning, Computer Vision, and Robotics as it relates to machine autonomy and medical applications.

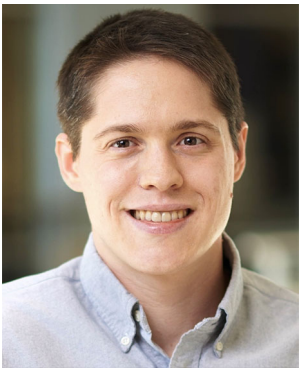
Naveen Vivek got a BSc in Electrical Engineering from Purdue University. He is currently a Product Development Engineer on the Yield Volume Diagnostics team at Advanced Micro Devices (AMD) Inc.

Abhinav Goel received his Ph.D. from Purdue University. He is currently a Senior Deep Learning Architect at NVIDIA. His research interests focus on optimizing the performance of the latest AI models to run faster.



Nicholas M. Synovic is an AI Application Architect and Computer Science Researcher at Loyola University Chicago where he conducts empirical research on the intersection of software engineering, AI, and deep neural network (DNN) reuse. He has achieved his Bachelor's and Master's degrees (in 2022 and 2024 respectively) in Computer Science from LUC.

George K. Thiruvathukal is Professor and Chair, Loyola University Chicago and Visiting Computer Scientist at Argonne National Laboratory in the Leadership Computing Facility. His research topics include parallel and distributed systems, software engineering, computer science, and embedded systems. He is a Senior Member of the IEEE.



James C. Davis is an assistant professor in the Elmore Family School of Electrical & Computer Engineering at Purdue University. His research interests include human and technical aspects of software engineering and cybersecurity, with an emphasis on the analysis of failures. Davis received his PhD in Computer Science from Virginia Tech. He is a Member of the ACM and a Senior Member of the IEEE. Contact him at davisjam@purdue.edu

Authors and Affiliations

Wenxin Jiang[1] · Vishnu Banna[1] · Naveen Vivek[1] · Abhinav Goel[1] · Nicholas Synovic[2] · George K. Thiruvathukal[2] · James C. Davis[1]

✉ James C. Davis
davisjam@purdue.edu

Wenxin Jiang
jiang784@purdue.edu

Vishnu Banna
vbanna@purdue.edu

Naveen Vivek
vivek@purdue.edu

Abhinav Goel
goel39@purdue.edu

Nicholas Synovic
nsynovic@luc.edu

George K. Thiruvathukal
gkt@cs.luc.edu

[1] Purdue University, West Lafayette, IN, USA

[2] Loyola University Chicago, Chicago, IL, USA