Compatibility Issues in Deep Learning Systems: Problems and Opportunities

Jun Wang
junwang@nuaa.edu.cn
Nanjing University of Aeronautics and Astronautics, China

Guanping Xiao*
gpxiao@nuaa.edu.cn
Nanjing University of Aeronautics and Astronautics, China

Shuai Zhang
shuaizhang@nuaa.edu.cn
Nanjing University of Aeronautics and Astronautics, China

Huashan Lei
leihuashan@nuaa.edu.cn
Nanjing University of Aeronautics and Astronautics, China

Yepang Liu†
liuyp1@sustech.edu.cn
Southern University of Science and Technology, China

Yulei Sui
y.sui@unsw.edu.au
University of New South Wales, Australia

ABSTRACT

Deep learning (DL) systems are complex component-based systems, which consist of core program (code implementation and data), Python (language and interpreter), third-party libraries, low-level libraries, development tools, OS, and hardware environments. Incompatible interaction between components would cause serious compatibility issues, substantially affecting the development and deployment processes. What types of compatibility issues are frequently exposed in DL systems? What are the root causes of such issues and how do developers fix them? How far are we from automatically detecting and fixing DL compatibility issues? Although there are many existing studies on DL bugs, the characteristics of DL compatibility issues have rarely been systematically studied and the above questions remain largely unexplored. To fill this gap, we conduct the first comprehensive empirical study to characterize compatibility issues in DL systems. Through analyzing 352 DL compatibility issues classified from 3,072 posts on Stack Overflow, we present their types, manifestation stages, and symptoms. We further summarize the root causes and common fixing strategies, and conduct a tool survey on the current research status of automated detection and repair of DL compatibility issues. Our study allows researchers and practitioners to gain a better understanding of DL compatibility issues and can facilitate future tool development.

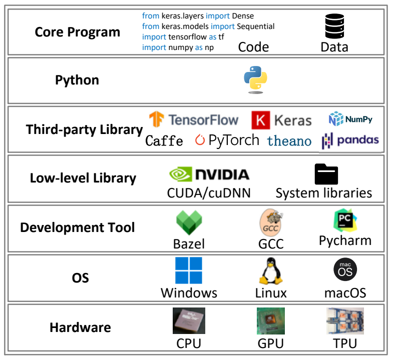
CCS CONCEPTS

• **Software and its engineering → Software defect analysis**; **Software libraries and repositories**; • **General and reference → Empirical studies**.

KEYWORDS

deep learning, compatibility issues, empirical study

ACM Reference Format:
Jun Wang, Guanping Xiao, Shuai Zhang, Huashan Lei, Yepang Liu, and Yulei Sui. 2023. Compatibility Issues in Deep Learning Systems: Problems and Opportunities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616321



```
from keras.layers import Dense
from keras.models import Sequential
import tensorflow as tf
import numpy as np
```

Figure 1: Components of DL Systems

1 INTRODUCTION

With the continuous evolution of deep learning (DL) technology, numerous DL-related software systems have been ubiquitously produced and deployed, such as medical imaging [62], autonomous driving cars [83], and various software engineering tasks [47, 98, 112, 118, 132]. Like traditional software [66, 82, 101, 116], developing and deploying DL systems also face different types of compatibility issues, which tend to trigger unexpected problems, significantly obstructing the development and deployment processes.

As shown in Figure 1, DL systems are composed of several complex and interdependent components, including core program (code implementation and data), programming language (mainly referred to Python and its interpreter), third-party libraries (e.g., TensorFlow, PyTorch, Numpy, and Pandas), low-level libraries (e.g., CUDA, cuDNN, drivers, and many system libraries), development tools (e.g., GCC, Bazel, and PyCharm), operating systems (OS, e.g., Windows, Linux, and macOS), and hardware environments (e.g., CPU, GPU, and TPU). To satisfy the requirements of different software and

*Guanping Xiao is the corresponding author.
†Yepang Liu is affiliated with the Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems at SUSTech.

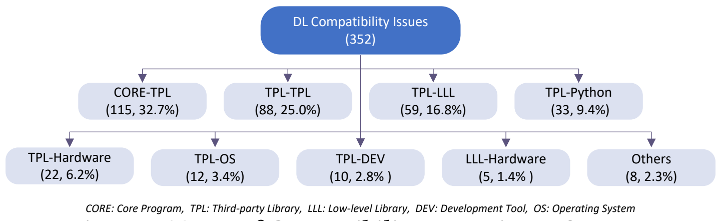


Figure 2: Types of Compatibility Issues in DL Systems

hardware environments, considering their frequently evolving versions, such components are specific and often incompatible with each other. Thus, compatible component-to-component interactions must be specified in accordance with certain constraints to make DL systems run normally.

On the one hand, APIs in third-party libraries are frequently changing due to fixing bugs, adding new features, and optimizing code implementation [43]. These coding activities may bring serious backward-breaking changes. One typical case is the change related to API usage, which offers an interface for developers to directly use a library feature in building core programs. Using removed or renamed APIs (e.g., function getting removed or renamed) in the program will inevitably cause the collapse of DL systems [129].

On the other hand, DL systems can only be built upon specific versions of components. Incompatible version combinations will cause unexpected issues during the installation or execution of DL systems. For example, according to the official installation guide [102], the low-level library for performing GPU computing of TensorFlow-GPU 1.2 should be CUDA 8. If users choose CUDA 7.5, no error message will appear in the TensorFlow installation stage. However, the system will throw an *ImportError* during execution, since the corresponding runtime CUDA file cannot be found, as reported in a Stack Overflow (SO) post [10]. In addition, due to API-breaking changes that occurred in the evolution of libraries, the interaction between third-party libraries should also rely on specific combinations of versions [124].

Although compatibility issues are commonly observed in DL systems, these issues are often challenging to locate and fix, especially for the one who is unfamiliar with the interaction between different components. In particular, when the system throws an exception at runtime, it is difficult for users to determine whether the root cause is induced by the incompatibility between software and hardware environments, the version mismatch between third-party libraries, or the breaking-changed API of third-party libraries used in the core program.

For example, another SO post reported that a developer tried to perform the object detection finetuning tutorial from PyTorch on a Linux laptop with a Nvidia GPU, but kept getting the *TypeError* [27], i.e., *object of type <class 'numpy.float64'> cannot be safely interpreted as an integer*. Strangely, the error did not happen on his home computer with the same OS and GPU environment a day before. He spent about 15 hours investigating the root cause and finally realized that in a recent release (i.e., Numpy 1.18.0), the `num` parameter in `numpy.linspace()` no longer accepts the float type. Hence, anyone who uses *pycocotools* API with Numpy 1.18.0 will encounter the same error. Simply downgrading the Numpy version to 1.17.4 solves this issue.

Therefore, it is practically valuable to study the characteristics of DL compatibility issues. Since DL systems have become more and more popular and important nowadays, there are many studies related to different comprehensive taxonomies of DL bugs [38, 39, 45, 46, 67, 68, 70, 122, 125, 127]. However, none of these studies specifically focus on the analysis of DL compatibility issues. Besides, existing efforts have primarily concentrated on analyzing the API evolution of Python libraries [44, 91, 128, 129] and resolving dependency conflicts in Python programs [42, 64, 84, 111]. For example, Zhang *et al.* [129] analyzed the evolution patterns of six Python framework APIs (i.e., TensorFlow, Keras, Scikit-learn, Pandas, Flask, and Django) and compatibility issues caused by breaking-changed APIs. Ye *et al.* [124] proposed a tool called PyEGo that aims to automatically infer compatible versions of Python, third-party libraries, and system libraries. However, these studies focus on a specific aspect of DL compatibility issues, but lack a comprehensive understanding of the prevalent types, symptoms, root causes, and fixing solutions of compatibility issues in DL systems.

To bridge this knowledge gap, in this paper, we conduct systematic study on compatibility issues in DL systems. First, we collect 3,072 posts from the official SO data dump using five tags (i.e., tensorflow, keras, pytorch, caffe, and theano) related to the five most discussed DL libraries on SO and 12 keywords (i.e., compatibility, version, evolution, exception, typeerror, attributeerror, importerror, modulenotfounderror, runtimeerror, compatible, cuda, cudnn). Then, we perform manual classification on these posts to identify DL compatibility issues (i.e., 352) and further investigate their characteristics. Last, we conduct a tool survey to study the current state of relevant research on automated detection and repair of DL compatibility issues. Our work mainly focuses on answering the following three research questions (RQs).

- **RQ1.** What types of compatibility issues are frequently exposed in DL systems?
- **RQ2.** What are the root causes of DL compatibility issues and how do developers fix them?
- **RQ3.** How far are we from automatically detecting and fixing DL compatibility issues?

In this paper, we make the following contributions:

- **Definition.** A list of components (including core program, Python, third-party library, low-level library, development tool, OS, and hardware) in DL systems, and the definition of DL compatibility issues (Section 2).
- **Empirical Study.** We conduct the first empirical study of DL compatibility issues by analyzing 352 issues classified from 3,072 SO posts (Sections 3 and 4).
- **Tool Survey.** We survey 15 tools relevant to detecting and fixing DL compatibility issues selected from the most recent editions (18-22) of the top SE conferences, i.e., ICSE, FSE, and ASE (Section 5).

- **Implications and Suggestions.** We compare compatibility issues in DL systems with those in traditional software systems and provide implications and suggestions for researchers and practitioners based on our empirical findings (Section 6).
- **Datasets.** We release the replication dataset with a detailed instruction document of DL compatibility issues (Section 10).

2 DL COMPATIBILITY ISSUES

DL System: In this study, we define a *DL system* as the composition of seven interdependent components, including *core program*, *Python*, *third-party library*, *low-level library*, *development tool*, *OS*, and *hardware*, as depicted in Figure 1. Each component is briefly described as follows:

(1) *Core Program (CORE):* The core program (CORE) consists of code implementation and data. Code implementation mainly includes data processing, model definition, hyper-parameter settings, and the optimization training process [88]. Data stands for the datasets used in model development and deployment.

(2) *Python:* Python, the programming language and its interpreter environment, provides the syntax and runtime environment to implement and execute the core program. For example, the core program can be written in Python 2 or Python 3. In this study, we consider Python as the major programming language used for implementing DL systems.

(3) *Third-party Library (TPL):* TPLs provide predefined functions through APIs, which are used by developers to implement corresponding features in the core program as well as other TPLs. For example, DL frameworks (e.g., TensorFlow and PyTorch) provide the features to build and optimize DL models. Numpy and Pandas offer several data processing functions. Using APIs from TPLs can simplify and accelerate software development [109].

(4) *Low-level Library (LLL):* LLLs provide APIs to access hardware and system resources, which are fundamental parts required by several TPLs. For example, the GPU computing operations of TensorFlow are performed through CUDA/cuDNN libraries [88]. Besides, system libraries (e.g., Glibc and Glibcxx) provide standard C/C++ APIs and OS-specific APIs for TPLs, since the low-level implementation of many TPLs was written in C/C++ [124].

(5) *Development Tool (DEV):* DEVs are the tools used in development and deployment processes, usually related to building/compiling, executing activities, etc. For example, Bazel is the official tool for building TensorFlow from source [36].

(6) *Operating System (OS):* OS provides the software computing environment. Common OSs in DL systems include Windows, Linux, and macOS.

(7) *Hardware:* Hardware provides the hardware computing environment for DL systems [119]. Typical computing devices include CPU, GPU, and TPU.

DL Compatibility Issue: Based on the above definition, we define the DL compatibility issue as the incompatible interaction problem between components in the DL system.

3 METHODOLOGY

3.1 Data Collection

3.1.1 Stack Overflow Data. To study compatibility issues in DL systems, we have collected data from Stack Overflow (SO), one of the most popular question-answering sites concentrating on programming-related questions [121]. Posts on SO are frequently updated. This may bring potential threats to the reliability of our findings. Thus, we used the official stack exchange data dump released at *archive.org* [33], which provides each separate part of the whole SO website, including *Posts*, *Users*, *Votes*, *Comments*, *PostHistory*, *PostLinks*, *Tags*, and *PostLinks.*

The latest SO dump data was published on June 6, 2022, when we started this work, and later updated on October 5, 2022. To guarantee questions reflect the latest issues that DL developers and users encountered, we used the data released on October 5, 2022. In our study, we downloaded *stackoverflow.com-Posts.7z* and *stackoverflow.com-Tags.7z*, which contain *Posts.xml* and *Tags.xml* respectively, as the raw data.

3.1.2 Data Collection Process. There are 23,020,127 question posts in the collected SO dump data, spanning from July 2008 to October 2022. We defined the following five rules as the criteria to sample a small set as our classification data:

Rule 1. The post is a question post. The field `PostTypeId` in *Posts.xml* represents the type of a post: 1 for a question post and 2 for an answer post [121].

Rule 2. The post is still open for discussion. Some posts are labeled as closed due to duplication, needing details or clarification, or out of topic. These posts may not have sufficient information for our analysis. To ensure the reliability of our study, we discarded such closed posts.

Rule 3. The post has an accepted answer and the score (computed by upvotes minus downvotes) of the post is no less than 3. Posts with accepted answers indicate that the issues have been resolved, as recognized by the original posters [41, 67]. To further ensure the quality and quantity of posts [68], we set a minimum score threshold of 3.

Rule 4. The post contains at least one of the five tags, i.e., *tensorflow*, *keras*, *pytorch*, *caffe*, and *theano*. These five tags represent the five most discussed DL libraries on SO, according to the number of related posts.

Rule 5. The post's title or body contains at least one of the 12 keywords, i.e., *compatibility*, *version*, *evolution*, *exception*, *typeerror*, *attributeerror*, *importerror*, *modulenotfounderror*, *runtimeerror*, *compatible*, *cuda*, and *cudnn*. The first seven keywords are adopted from [129], where the authors used them to search for compatibility issues in GitHub related to the API evolution of Python TPLs. Based on our domain knowledge in developing and testing DL projects, we extend the keyword set by adding five new keywords, i.e., *modulenotfounderror*, *runtimeerror*, *compatible*, *cuda*, *cudnn.*

Based on the above filtering conditions, we developed Python scripts to automatically extract tags and posts from *Tags.xml* and *Posts.xml* files. Finally, 3,072 posts are collected for further manual classification, as shown in Table 1[1].

3.2 DL Compatibility Issues Classification

To identify and characterize DL compatibility issues, we performed an iterative manual labeling process by following the widely-adopted

[1]The sum of the number of posts in each tag exceeds 3,072 because a post can have multiple tags.

Table 1: Summary of Dataset in Our Study

Tag	*TensorFlow*	*Keras*	*PyTorch*	*Caffe*	*Theano*
# Posts	1,999	982	592	67	127
# Comp. Posts	265	117	33	9	16

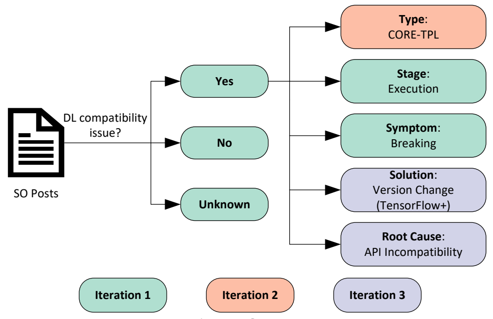


Figure 3: Classification Process

open coding procedure [107], as shown in Figure 3. Two annotators (i.e., co-authors of this paper), who are all familiar with DL project development, spent five months on the classification. In each iteration, a third author participated in the discussion to solve the inconsistencies and finally reached a consensus. To measure the inter-rater agreement between two annotators, we calculated Cohen's kappa coefficient for each iteration [104]. The iterations are as follows:

Iteration 1. For the 3,072 posts, we analyzed three parts in a post, i.e., the question descriptions (including code snippets, exception information (e.g., traceback) and the attached external links), the answers in the post, and the comments discussed by users. Two annotators independently inspected these sources of data on the SO website, which has better readability than the *Posts.xml*. Since the online post may be further updated, we compared the last modified dates between the online post and the extracted one from *Posts.xml* to ensure consistency. If the dates are different, we ignore the new information (e.g., new answers or comments) posted on the website after the last modified date recorded in *Posts.xml*.

In this iteration, we first checked whether a post discusses a DL compatibility issue, according to the definitions described in Section 2. Specifically, we focused on the information related to the version change of components, such as TPLs, LLLs, and Python. If there is no such information, we further checked whether the issue was induced by API evolution.

If a post is labeled as ***Yes***, we further described corresponding counterparts of the issues, i.e., the incompatible problem was induced between which components, the manifestation stage of such issue, and the consequent impact on the DL system. If the poster did not specify the version of the incompatible component, we can infer the range of versions prior to the post submission time by examining the release history on GitHub and the Python-PyPI repository. It is noted that the library in an issue could be different from the tag attached to the post since the tag is a general label related to the discussion topic on SO. Besides, we also recorded the solution posted by the accepted answer and described the root cause based on our own understanding. If posts do not contain sufficient information to be classified as a compatibility issue or not, we label them as ***Unknown***. Posts related to API misuse and usage query, general questions asking for code implementation to a specific function, and environment configuration issues related to missing required dependencies caused by forgetting to configure/install necessary components but not due to version mismatch, are labeled as ***No***.

We then cross-checked the labeling results and mainly compared the identification of compatibility issues and their stage and impact. The value of Cohen's kappa coefficient is 0.89 in this iteration.

Iteration 2. In the second iteration, we independently relabeled all posts based on the preliminary classification from the first round. In particular, we focused on the type of DL compatibility issues, i.e., to determine which two components introduce the issue. To further understand and identify the type, we created virtual environments using Conda [32], trying to reproduce the reported issues, according to the given versions of Python and TPLs. For the compatibility issues related to CORE and TPL, we used *git blame* to identify the commit on GitHub that introduced the API changes and further labeled the API evolution pattern by adopting the taxonomy from [44]. Note that the reproduction process helps us to match the incompatible library version with the specific breaking changes that were introduced. For example, the code snippet runs normally in version *V1*, but crashes in the subsequent version *V2*. By reproducing, we updated the labeling results of DL compatibility issues, solutions, and root causes with more detailed information. Misclassified posts in the first iteration can be further verified. The value of Cohen's kappa coefficient in this iteration is 0.92.

Iteration 3. In the last round, we focused on the labeling of solutions and root causes of DL compatibility issues. Specifically, we further checked whether the proposed solution is effective according to discussions posted by users and our reproducing results. Since posts could have several answers, we concentrated on analyzing those, whose submitted time is close to the issue posted time. Due to the evolving versions of components, new answers (solutions) are often not appropriate for the original issue, which was posted years ago. We finally summarized the solutions and root causes of the DL compatibility issues. The value of Cohen's kappa coefficient is 0.95 for the last iteration.

After three iterations, we classified 352 DL compatibility issues from 3,072 SO posts. Table 1 depicts the distribution of the 352 issues in the set of SO posts. Note that the sum of the number of posts is greater than 352, as a post can have more than one tag.

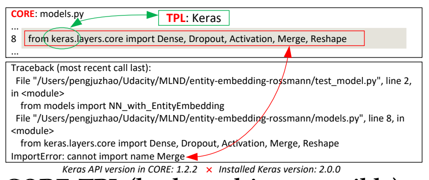
Table 2: Distribution of Stages and Types

Type	Installation	Execution	Total
CORE-TPL	0	115	115
TPL-TPL	5	83	88
TPL-LLL	3	56	59
TPL-Python	28	5	33
TPL-Hardware	1	21	22
TPL-OS	6	6	12
TPL-DEV	6	4	10
LLL-Hardware	2	3	5
Others	1	7	8
Total	52	300	352

4 ANALYSIS RESULTS

4.1 RQ1: Types, Stages and Symptoms

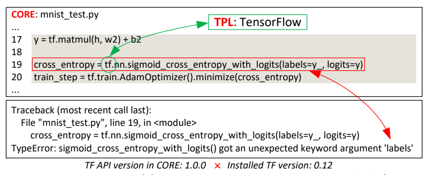
4.1.1 Types. Based on the classification process described above, we classified the 352 compatibility issues in DL systems into nine types, including *CORE-TPL*, *TPL-TPL*, *TPL-LLL*, *TPL-Python*, *TPL-Hardware*, *TPL-OS*, *TPL-DEV*, *LLL-Hardware*, and *others*, as shown in Figure 2. Each issue is labeled as one leaf category of our classification taxonomy. Below, we will discuss each type in detail.


TPL: Keras
Keras API version in CORE: 1.2.2 × Installed Keras version: 2.0.0

```
CORE: models.py
...
8   from keras.layers.core import Dense, Dropout, Activation, Merge, Reshape
...
```

```
Traceback (most recent call last):
  File "/Users/pengjuzhao/Udacity/MLND/entity-embedding-rossmann/test_model.py", line 2,
in <module>
    from models import NN_with_EntityEmbedding
  File "/Users/pengjuzhao/Udacity/MLND/entity-embedding-rossmann/models.py", line 8, in
<module>
    from keras.layers.core import Dense, Dropout, Activation, Merge, Reshape
ImportError: cannot import name Merge
```

Figure 4: CORE-TPL (backward-incompatible): #42823627



```
CORE: mnist_test.py
...
17  y = tf.matmul(h, w2) + b2
18
19  cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(labels=y_, logits=y)
20  train_step = tf.train.AdamOptimizer().minimize(cross_entropy)
...
```

```
Traceback (most recent call last):
   File "mnist_test.py", line 19, in <module>
      cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(labels=y_, logits=y)
TypeError: sigmoid_cross_entropy_with_logits() got an unexpected keyword argument 'labels'
```

Figure 5: CORE-TPL (forward-incompatible): #42675391

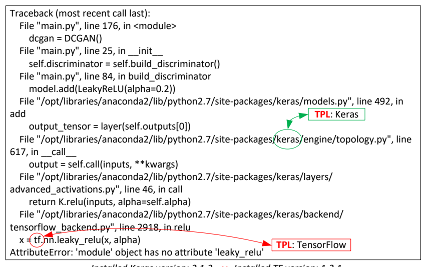
(1) *Core Program and Third-party Library (CORE-TPL):* This is the most common type (115/32.7%) of DL compatibility issues. Developers usually use APIs from a specific version of TPLs to implement the core program. If the installed version is different from the used one, compatibility issues are likely to occur, due to API evolution [129]. According to a newer or older installed version than the used one, *CORE-TPL* issues can be further classified as ***backward-incompatible*** (69) and ***forward-incompatible*** (46).

For example, Figure 4 shows a backward-incompatible issue [8]. The error occurred when importing class `Merge` from *keras.layers.core*. This is because class `Merge` has been moved to another module since version 2.0 [50]. The same import statement can be executed under version 1.2.2. On the contrary, Figure 5 shows a forward-incompatible issue [7], in which an error appeared when calling TensorFlow API `tf.nn.sigmoid_cross_entropy_with_logits()` with parameter `labels` in the core program. The parameter `labels` was added in TensorFlow 1.0.0 [49], while the developer installed an older version (e.g., 0.12). In addition, we found that the top five TPLs of *CORE-TPL* issues are TensorFlow (63), Keras (34), PyTorch (6), Theano (3), and TorchText (2).

Finding 1: *CORE-TPL* (32.7%) is the most frequent type of DL compatibility issues. 60.0% of *CORE-TPL* issues are backward-incompatible issues, and the remaining 40.0% are forward-incompatible issues. Most (84.3%) of *CORE-TPL* issues are related to TensorFlow (54.8%) and Keras (29.5%).

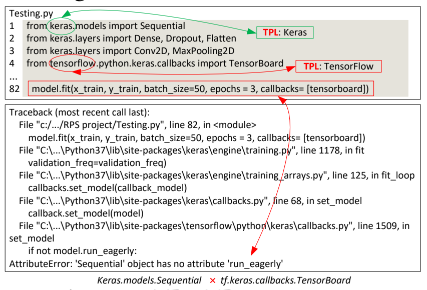
(2) *Third-party Library and Third-party Library (TPL-TPL):* is the second most frequent type (88/25.0%). Developing Python TPLs also relies on several other TPLs. The triggering factor of *TPL-TPL* mainly lies in the installation of TPLs with incompatible versions. Since TPLs are frequently evolving [91], it is often difficult for users to choose compatible versions, especially when the constraint condition (e.g., `requirements.txt`) is not provided [124]. We found the top four related TPLs of *TPL-TPL* issues are TensorFlow (67), Keras (42), Numpy (9), and Theano (5). In particular, *TensorFlow-Keras* introduces the most issues (37), as they have a strong version constraint and a large user population.

Keras provides user-friendly API for building neural networks and supports multi backends [68], e.g., TensorFlow and Theano. For example, Figure 6 shows a compatibility issue induced by incompatible versions of Keras and TensorFlow. The exception occurred inside Keras when invoking TensorFlow API `tf.nn.leaky_relu()`


TPL: Keras
TPL: TensorFlow
Installed Keras version: 2.1.3 × Installed TF version: 1.2.1

```
Traceback (most recent call last):
   File "main.py", line 176, in <module>
      dcgan = DCGAN()
   File "main.py", line 25, in __init__
      self.discriminator = self.build_discriminator()
   File "main.py", line 84, in build_discriminator
      model.add(LeakyReLU(alpha=0.2))
   File "/opt/libraries/anaconda2/lib/python2.7/site-packages/keras/models.py", line 492, in
add
      output_tensor = layer(self.outputs[0])
   File "/opt/libraries/anaconda2/lib/python2.7/site-packages/keras/engine/topology.py", line
617, in __call__
      output = self.call(inputs, **kwargs)
   File "/opt/libraries/anaconda2/lib/python2.7/site-packages/keras/layers/
advanced_activations.py", line 46, in call
      return K.relu(inputs, alpha=self.alpha)
   File "/opt/libraries/anaconda2/lib/python2.7/site-packages/keras/backend/
tensorflow_backend.py", line 2918, in relu
   x = tf.nn.leaky_relu(x, alpha)
AttributeError: 'module' object has no attribute 'leaky_relu'
```

Figure 6: TPL-TPL: #48929098



```
Testing.py
1   from keras.models import Sequential
2   from keras.layers import Dense, Dropout, Flatten
3   from keras.layers import Conv2D, MaxPooling2D
4   from tensorflow.python.keras.callbacks import TensorBoard
...
82  model.fit(x_train, y_train, batch_size=50, epochs = 3, callbacks= [tensorboard])
```

```
Traceback (most recent call last):
   File "c:/.../RPS project/Testing.py", line 82, in <module>
      model.fit(x_train, y_train, batch_size=50, epochs = 3, callbacks= [tensorboard])
   File "C:\...\Python37\lib\site-packages\keras\engine\training.py", line 1178, in fit
      validation_freq=validation_freq)
   File "C:\...\Python37\lib\site-packages\keras\engine\training_arrays.py", line 125, in fit_loop
      callbacks.set_model(callback_model)
   File "C:\...\Python37\lib\site-packages\keras\callbacks.py", line 68, in set_model
      callback.set_model(model)
   File "C:\...\Python37\lib\site-packages\tensorflow\python\keras\callbacks.py", line 1509, in
set_model
      if not model.run_eagerly:
AttributeError: 'Sequential' object has no attribute 'run_eagerly'
```

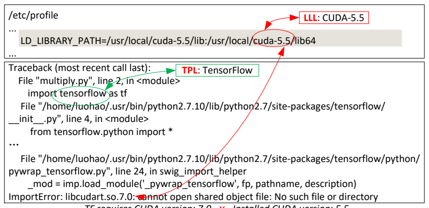
Figure 7: TPL-TPL: #57718512

in the implementation of layer `LeakyReLU` [12]. The developer installed TensorFlow 1.2.1 and Keras 2.1.3. However, the function `leaky_relu` was added in TensorFlow 1.4 [51], resulting the *AttributeError*. Besides, the mixed use of APIs from Keras and tf.keras (i.e., the integrated Keras APIs in TensorFlow) could also bring compatibility issues [24], as shown in Figure 7. Keras and tf.keras are not compatible, as the implementation of tf.keras is not identical to the standalone Keras [22, 23, 28].

Finding 2: *TPL-TPL* (25.0%) is the second most common type of DL compatibility issues. 76.1% of *TPL-TPL* issues are induced between TensorFlow and other TPLs, of which Keras accounts for 55.2%.

(3) *Third-party Library and Low-level Library (TPL-LLL):* 59 issues are caused by incompatible versions between TPLs and LLLs, accounting for 16.8%. Among them, 41 issues are related to TPL-CUDA/cuDNN. In particular, TensorFlow and CUDA, TensorFlow and cuDNN introduce 25 and 15 issues, respectively. One issue is related to both. Unlike PyTorch which integrates CUDA/cuDNN libraries in its installation wheel [29], using GPU for accelerating DL model training by TensorFlow requires users to install and configure respective compatible CUDA/cuDNN versions according to the TensorFlow versions [102]. For example, Figures 8 and 9 illustrate two typical compatibility issues related to TensorFlow and CUDA/cuDNN. During runtime, TensorFlow throws an *ImportError* since `libcudart.so.7.0` can not be found [3]. This implies that TensorFlow requires CUDA 7 to perform GPU computing, but the installed CUDA version is 5.5. Besides, the cuDNN issue depicted in Figure 9 describes that the version of cuDNN library that TensorFlow compiled against (i.e., 5.1.3) is incompatible with the configured cuDNN library (i.e., 5.0.5) at runtime [5].

Finding 3: *TPL-LLL* (16.8%) is the third most frequent type of DL compatibility issues. 69.5% of LLLs are CUDA and cuDNN, while the remaining 30.5% are related to Nvidia GPU drivers and system libraries (e.g., glibc, libstdc++, and boost).

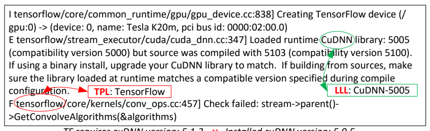


```
/etc/profile
...
  LD_LIBRARY_PATH=/usr/local/cuda-5.5/lib:/usr/local/cuda-5.5/lib64
...
```

```
Traceback (most recent call last):
  File "multiply.py", line 2, in <module>
    import tensorflow as tf
   File "/home/luohao/.usr/bin/python2.7.10/lib/python2.7/site-packages/tensorflow/
__init__.py", line 4, in <module>
     from tensorflow.python import *
...
   File "/home/luohao/.usr/bin/python2.7.10/lib/python2.7/site-packages/tensorflow/python/
pywrap_tensorflow.py", line 24, in swig_import_helper
    _mod = imp.load_module('_pywrap_tensorflow', fp, pathname, description)
ImportError: libcudart.so.7.0: cannot open shared object file: No such file or directory
```

TF requires CUDA version: 7.0 × *Installed CUDA version: 5.5*

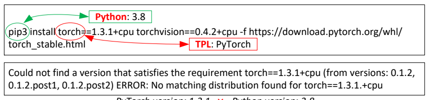
Figure 8: TPL-LLL: #33671372



```
I tensorflow/core/common_runtime/gpu/gpu_device.cc:838] Creating TensorFlow device (/
gpu:0) -> (device: 0, name: Tesla K20m, pci bus id: 0000:02:00.0)
E tensorflow/stream_executor/cuda/cuda_dnn.cc:347] Loaded runtime CuDNN library: 5005
(compatibility version 5000) but source was compiled with 5103 (compatibility version 5100).
If using a binary install, upgrade your CuDNN library to match.  If building from sources, make
sure the library loaded at runtime matches a compatible version specified during compile
configuration.
F tensorflow/core/kernels/conv_ops.cc:457] Check failed: stream->parent()-
>GetConvolveAlgorithms(&algorithms)
```

TF requires cuDNN version: 5.1.3 × *Installed cuDNN version: 5.0.5*

Figure 9: TPL-LLL: #41005249

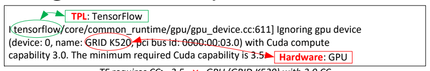


```
pip3 install torch==1.3.1+cpu torchvision==0.4.2+cpu -f https://download.pytorch.org/whl/
torch_stable.html
```

```
Could not find a version that satisfies the requirement torch==1.3.1+cpu (from versions: 0.1.2,
0.1.2.post1, 0.1.2.post2) ERROR: No matching distribution found for torch==1.3.1.+cpu
```

PyTorch version: 1.3.1 × *Python version: 3.8*

Figure 10: TPL-Python: #58901682



```
I tensorflow/core/common_runtime/gpu/gpu_device.cc:611] Ignoring gpu device
(device: 0, name: GRID K520, pci bus id: 0000:00:03.0) with Cuda compute
capability 3.0. The minimum required Cuda capability is 3.5.
```

TF requires CC>=3.5 × *GPU (GRID K520) with 3.0 CC*

Figure 11: TPL-Hardware: #33651810

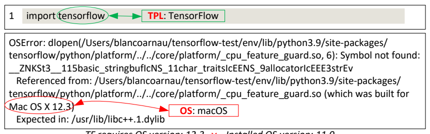
(4) *Third-party Library and Python (TPL-Python):* There are 33 DL compatibility issues caused by incompatibility between TPL and Python, accounting for 9.4%. For example, Figure 10 shows that the developer trying to use `pip` to install PyTorch [25]. However, there are no PyTorch wheels for Python 3.8 at the posted time, resulting in `pip` being unable to find a compatible distribution for PyTorch. In addition, using 32-bit Python to install 64-bit TPL wheels also lead to compatibility issues [9].

Finding 4: 9.4% of DL compatibility issues are related to *TPL-Python*.

(5) *Third-party Library and Hardware (TPL-Hardware):* Among the 352 issues, 22 issues are caused by the incompatibility between TPL and hardware, accounting for 6.2%. For example, Figure 11 shows that the minimum CUDA capability required by TensorFlow is 3.5 while the CUDA capability of GRID K520 GPU is 3.0 [2]. As a result, the GPU device will be ignored and the model training will be performed on CPU. The hardware types are CPU (11), GPU (8), TPU (2), and Raspberry Pi (1), while TPLs are related to TensorFlow (18), PyTorch (2), Theano (1), and Lasagne (1). Moreover, 50.0% (11) of *TPL-Hardware* issues are related to the incompatibility between TensorFlow and CPU. This is mainly because starting with version 1.6, TensorFlow binaries were built with enabling the support of AVX instructions by default [53], which may not run on older CPU, e.g., Intel(R) Core(TM) 2 Duo CPU T5870 [19], AMD Athlon Dual Core 4450e [13], and Intel(R) Pentium(R) 3556U [21].

Finding 5: *TPL-Hardware* accounts for 6.2% of DL compatibility issues. 50.0% of *TPL-Hardware* issues are caused by the incompatibility between TensorFlow and CPU.

(6) *Third-party Library and Operating System (TPL-OS):* 12 DL compatibility issues are *TPL-OS*, accounting for 3.4%. Common OS types in this category are macOS (5), Linux (3), Windows (3), and Raspian Lite OS (1). We can observe from the example shown in Figure 12 that, a symbol was missing in `_cpu_feature_guard.so`

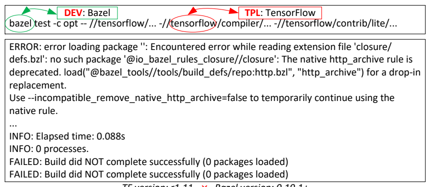


```
import tensorflow
```

```
OSError: dlopen(/Users/blancoarnau/tensorflow-test/env/lib/python3.9/site-packages/
tensorflow/python/platform/../../core/platform/_cpu_feature_guard.so, 6): Symbol not found:
__ZNKSt3__115basic_stringbufIcNS_11char_traitsIcEENS_9allocatorIcEEE3strEv
  Referenced from: /Users/blancoarnau/tensorflow-test/env/lib/python3.9/site-packages/
tensorflow/python/platform/../../core/platform/_cpu_feature_guard.so (which was built for
Mac OS X 12.3)
  Expected in: /usr/lib/libc++.1.dylib
```

TF requires OS version: 12.3 × *Installed OS version: 11.0*

Figure 12: TPL-OS: #71174306



```
bazel test -c opt -- //tensorflow/... -//tensorflow/compiler/... -//tensorflow/contrib/lite/...
```

```
ERROR: error loading package '': Encountered error while reading extension file 'closure/
defs.bzl': no such package '@io_bazel_rules_closure//closure': The native http_archive rule is
deprecated. load("@bazel_tools//tools/build_defs/repo:http.bzl", "http_archive") for a drop-in
replacement.
Use --incompatible_remove_native_http_archive=false to temporarily continue using the
native rule.
...
INFO: Elapsed time: 0.088s
INFO: 0 processes.
FAILED: Build did NOT complete successfully (0 packages loaded)
FAILED: Build did NOT complete successfully (0 packages loaded)
```

TF version: r1.11 × *Bazel version: 0.19.1+*

Figure 13: TPL-DEV: #53707068

when importing TensorFlow on macOS (11.0) [31]. This is because the installed TensorFlow version was built for macOS 12.3.

Finding 6: 3.4% of DL compatibility issues are *TPL-OS*.

(7) *Third-party Library and Development Tool (TPL-DEV):* There are 10 compatibility issues are classified as *TPL-DEV*, accounting for 2.8%. The respective development tools include compiler/building tools (i.e., Bazel (4), GCC (2), and G++ (1)), IDE (i.e., Xcode (1) and PySpark (1)), and other (e.g., broswer (1)). For example, Figure 13 shows a *TPL-DEV* issue related to Bazel [17]. The developer attempted to build TensorFlow r1.11 from source using Bazel but failed, since some Bazel features that TensorFlow used have been deprecated in Bazel versions newer than 0.18.1 (e.g., 0.19.1).

Finding 7: 2.8% of DL compatibility issues are related to *TPL-DEV*.

(8) *Low-level Library and Hardware (LLL-Hardware):* Five issues are classified as *LLL-Hardware*, accounting for 1.4%. This category mainly includes incompatibility between CUDA/cuDNN and GPU. For example, CUDA 9 does not support GPU architecture `compute_20` and `sm_20` [11]. Nvidia Fermi M2090 is a GPU with compute capability 2.0 that does not support cuDNN library [1].

Finding 8: 1.4% of DL compatibility issues are *LLL-Hardware*.

(9) *Others:* Each type in this category has no more than five issues, which are related to *CORE-Python* (2), *LLL-LLL* (4), *OS-Hardware* (1), and *DEV-Hardware* (1). For example, posts with the IDs #38546672 [4], #55261785 [20], #51320027 [14], and #69865825 [30] for the four types, respectively.

Finding 9: DL compatibility issues can be also related to *CORE-Python*, *LLL-LLL*, *OS-Hardware*, and *DEV-Hardware*.

Figure 14 shows the evolution of DL compatibility issues. Over time, all types of DL compatibility issues show an increasing trend. Among these types, CORE-TPL, TPL-TPL, and TPL-LLL have received more attention from developers. With the evolution of DL frameworks, API signatures and behaviors inevitably change to accommodate new requirements, resulting in incompatible APIs (CORE-TPL). Besides, the integration of different TPLs when building DL systems often leads to version mismatches between the libraries involved, causing more TPL-TPL issues. Moreover, TPL-LLL issues have become more prevalent because DL systems often require DL frameworks to interact with low-level libraries such

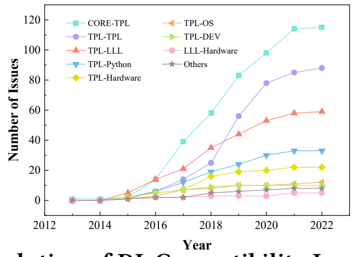


Figure 14: Evolution of DL Compatibility Issues over Time

as CUDA and cuDNN. This interaction is necessary to improve training efficiency, but it can introduce compatibility issues.

4.1.2 Stages. We classified the manifestation stage of DL compatibility issues into two stages, as depicted in Table 2:

- **Installation.** The issue appeared in the process of the environment configuration of DL systems, such as selecting software and hardware environments, and installing/compiling TPLs.
- **Execution.** The issue that appeared in the execution/running of DL systems.

We observed that 300 issues appeared in the execution stage, while only 52 issues were found in the installation stage. This indicates that most DL compatibility issues are not explicit, and often expose only at runtime. In particular, most of *CORE-TPL* (100%), *TPL-TPL* (94.3%), *TPL-LLL* (94.9%), and *TPL-Hardware* (95.5%) occurred during execution. In most cases, compatibility issues would occur when CORE/TPL dynamically invokes other TPLs' APIs.

On the contrary, most of *TPL-Python* (84.8%) and *TPL-DEV* (60.0%) issues are manifested during the installation stage. For example, 28 *TPL-Python* issues are related to the installation stage. Given a TPL, its supported Python versions are often limited, especially for an older Python version (e.g., 2.7 [6]) or a recently released Python version (e.g., Figure 10). Moreover, the number of compatibility issues related to *TPL-OS* is the same during installation and execution, both of which are 6. For *LLL-Hardware*, three issues occurred in the execution stage, while other two occurred during installation.

> **Finding 10:** Most (85.2%) of DL compatibility issues were exposed during the execution stage. *CORE-TPL* (100%), *TPL-TPL* (94.3%), *TPL-LLL* (94.9%), and *TPL-Hardware* (95.5%) are prone to occur during execution, while *TPL-Python* (84.8%) and *TPL-DEV* (60.0%) are likely to appear in the installation.

4.1.3 Symptoms. We classified the impact of compatibility issues on DL systems into three categories:

- **Breaking.** The issue led to the failure of components at the installation stage or the termination of the program during execution.
- **Low Performance.** The issue caused a low execution efficiency.
- **Unexpected Behavior.** The issue resulted in unexpected results but the systems did not throw an exception during execution.

We found that 321 DL compatibility issues have a breaking impact on both the installation and execution stages. For example, Figure 12 shows that the system encounters an *OSError* during execution. The large proportion of breaking impact on DL systems shows the necessity and urgency to develop tools for detecting and repairing such issues [106, 124, 129].

Table 3: Distribution of Root Causes and Types

Type	API Incompatibility		UC	VMCRL	Total
	VMCL	VMLL			
CORE-TPL	115	0	0	0	115
TPL-TPL	0	77	6	5	88
TPL-LLL	0	0	13	46	59
TPL-Python	0	0	33	0	33
TPL-Hardware	0	0	22	0	22
TPL-OS	0	0	12	0	12
TPL-DEV	0	0	10	0	10
LLL-Hardware	0	0	5	0	5
Others	0	0	7	1	8
Total	115	77	108	52	352

Although there are only a few compatibility issues that cause low performance (28) and unexpected behavior (3) to DL systems, their impact is not negligible. For the example displayed in Figure 11, due to the *TPL-Hardware* issue, the system will perform model training on the CPU. This will significantly impact the model training efficiency. In addition, API evolution may lead to unexpected behavior in the code implementation. For example, `torch.arrange()` returns a tensor of type float for 0.4.0 while it returns a tensor of type long for 0.4.1 [16].

> **Finding 11:** Most (91.2%) of DL compatibility issues have a breaking impact on the installation stage or execution stage of DL systems.

4.2 RQ2: Root Causes and Solutions

4.2.1 Root Causes. We identified the following three root causes of DL compatibility issues, including *API incompatibility*, *unsupported component*, and *version mismatch between compiled and runtime libraries*. Table 3 depicts the distribution of root causes of DL compatibility issues across the types.

(1) *Root Cause 1: API Incompatibility.* The DL compatibility issue is caused by the API evolution breaking the forward and/or backward compatibility. We can observe from Table 3 that API incompatibility is the most common root cause that induced DL compatibility issues (192/352). According to the interaction between components, API incompatibility can be further divided into two subcategories: *version mismatch between CORE and libraries' APIs* (VMCL: 115/192) and *version mismatch between libraries' APIs* (VMLL: 77/192).

> **Finding 12:** API incompatibility is the most common root cause, accounting for 54.5%. Of which, 59.9% are caused by VMCL and 40.1% are induced by VMLL.

Version Mismatch Between CORE and Libraries' APIs (VMCL) (115/192). It can be seen from Table 3 that all the *CORE-TPL* issues are introduced by VMCL. We further analyzed the API evolution pattern (adopted from [44]) in these issues, as shown in Table 4. Note that in the table, we categorize the rename and relocated patterns as part of the remove pattern. Specifically, *RemoveAlias* (16.5%), *RemoveModule* (16.5%), *AddParameter* (9.6%), *RemoveFunction* (8.7%), *RemoveClass* (7.8%), and *AddFuntion* (7.0%) are the top six API evolution patterns, accounting for 66.1%. It is noted that *RemoveAlias* is mainly found in the issues related to TensorFlow. For example, in order to make APIs of TensorFlow 1.0 compatible with TensorFlow 2.0, developers usually remove the original alias and add a new one, such as changing `tf.get_variable` to `tf.compat.v1.get_variable` [26]. In addition, although developers usually attempted to not breaking backward compatibility when changing APIs, these API evolution changes introduced 38.3% forward-incompatible issues, as mentioned in Finding 1.

Table 4: API Evolution Patterns in VMCL-Issues

Pattern	Subpattern	# Issues	Percentage	Total
Class	AddClass	5	4.3%	17
	RemoveClass	9	7.8%	
	ChangeInheritance	3	2.6%	
Function	AddFunction	8	7.0%	24
	RemoveFunction	10	8.7%	
	ChangeReturnType	3	2.6%	
	ChangeReturnValue	3	2.6%	
Parameter	AddParameter	11	9.6%	23
	RemoveParameter	8	7.0%	
	AddParameterDefault	1	0.9%	
	MoveParameter	3	2.6%	
Module	AddModule	3	2.6%	22
	RemoveModule	19	16.5%	
Alias	AddAlias	7	6.1%	26
	RemoveAlias	19	16.5%	
Attribute	AddAttribute	2	1.7%	2
	RemoveAttribute	0	0.0%	
Others	AddException	1	0.9%	1
Total		115	100.0%	115

Table 5: Distribution of Exceptions in VMCL and VMLL

Exception	VMCL	Exception	VMLL
AttributeError	47	AttributeError	38
ImportError	27	ImportError	16
TypeError	23	TypeError	13
ModuleNotFoundError	8	RuntimeError	3
RuntimeError	2	ValueError	2
Others	8	NotImplementedError	2
Total	115	Others	3
		Total	77

Finding 13: The most common API evolution patterns related to VMCL issues are *RemoveAlias* (16.5%), *RemoveModule* (16.5%), *AddParameter* (9.6%), *RemoveFunction* (8.7%), *RemoveClass* (7.8%), and *AddFuntion* (7.0%).

Version Mismatch Between Libraries' APIs (VMLL) (77/192). Among the 192 issues, 40.1% are caused by VMLL, which is the major root cause of *TPL-TPL*. This is mainly due to the independent development and design of different libraries. The lack of standardized specifications and limited testing also contribute to the issues. Particularly, as mentioned in Finding 2, *TensorFlow-Keras* introduced the most *TPL-TPL* issues, which are caused by VMLL, as the examples shown in Figures 6 and 7. There are 17 *TPL-TPL* issues related to the mixed use of Keras and tf.keras. Since TensorFlow 2.0, Keras becomes the core module of TensorFlow [54]. To avoid the compatibility issues caused by the mixed use of Keras and tf.keras, it is recommended to import all Keras API from TensorFlow, i.e., using tf.keras [55].

Besides, we further counted the exception type of issues caused by API incompatibility (i.e., VMCL and VMLL), as shown in Table 5. We found that *AttributeError*, *ImportError*, and *TypeError* are the top three most frequent exception types in API incompatibility issues, which confirms a similar result from [129].

Finding 14: *AttrituteError*, *ImportError*, and *TypeError* are the top three exceptions caused by API incompatibility.

(2) *Root Cause 2: Unsupported Component (UC).* 108 compatibility issues are caused by using unsupported components to the installation and/or execution of DL systems. UC is the second most common root cause that exists in all the types of DL compatibility issues except *CORE-TPL*. We can observe from Table 3 that all the *TPL-Python*, *TPL-Hardware*, *TPL-OS*, *TPL-DEV*, *LLL-Hardware*, and *others* are caused by UC. Some components are only compatible for specific environments. For example, TensorFlow only supports 64-bit Python shown in Figure 10. Google TPU does not support some graph operations when using TensorFlow [15].

Finding 15: 30.7% of DL compatibility issues are caused by UC, which is the second most common root cause. Besides, UC is one of the root causes of all types except *CORE-TPL*.

Table 6: Distribution of Solutions to API Incompatibility

Solution		VMCL		VMLL	Total
		forward	backward		
Version Change		31	6	57	94
Code Change	Change API	9	40	1	50
	Change Import	8	22	19	49
	Change Build	0	0	0	0

Table 7: Distribution of Solutions to UC and VMCRL

Solution		UC	VMCRL	Total
Version Change		76	52	128
Code Change	Change API	1	0	1
	Change Import	0	0	0
	Change Build	15	0	15

(3) *Root Cause 3: Version Mismatch Between Compiled and Runtime Libraries (VMCRL).* There are 52 (14.8%) DL compatibility issues caused VMCRL. Among them, 46 issues are related to *TPL-LLL*, while among the rest, five issues are *TPL-TPL* and one issues are *LLL-LLL*. Compilation and execution are two separate processes, which are often conducted by different developers, especially for the releases of TPL binaries. Thus, when executing a compiled binary, it is difficult to configure required libraries, if there is no guidance provided.

Finding 16: 14.8% of DL compatibility issues are caused by VMCRL, of which 88.5% are related to *TPL-LLL*.

4.2.2 *Solutions.*

By analyzing the 352 issues, we identified the following two common fixing strategies that developers often use to solve DL compatibility issues, covering 330 issues. Note that some issues can be fixed by more than one solution.

- **Version Change.** The issue was fixed by changing the version of a component.
- **Code Change.** The issue was fixed by changing the code, mainly including changing API, changing the import statement, and changing the build file.

(1) *Solutions to Root Cause 1.* Table 6 shows the distribution of solutions to DL compatibility issues caused by API incompatibility. For VMCL, it can be observed that 31 forward-incompatible issues were solved by changing the TPL version, while 17 issues were repaired by changing the code (API and import statement). By contrast, 62 backward-incompatible issues were fixed by changing the code, especially the API usage. The results are expected. It is convenient to upgrade the installed version to the corresponding new version for the forward-incompatible issues. However, for the backward-incompatible issues, developers need to change the API usage if they want to use the new TPL version. Besides, for VMLL, 57 issues were fixed by changing the version, and 20 issues got a code fix. This is because changing the incompatible TPL version is easier than changing the code in a TPL for fixing such issues.

Finding 17: For compatibility issues caused by VMCL, 67.4% of forward-incompatible issues were fixed by changing the TPL version, while 89.9% of backward-incompatible issues were repaired by changing the code. In addition, 74.0% of issues caused by VMLL were fixed by version change.

(2) *Solutions to Root Causes 2 and 3.*

Table 7 displays the distribution of solutions to UC and VMCRL. 76 UC-caused issues were repaired by changing the component version, while 16 issues were fixed by a code change, especially changing the build file to rebuild/remake. Besides, all the VMCRL-caused issues were fixed by a version change. For example, the C

Table 8: Library API Evolution

Tool	Extract Used API	Extract TPL API	Match API	Detect Changes	Repair API
AexPy [44]		✓		✓	
PyCompat [129]	✓	✓	✓	✓	
DLocator [109]	✓		✓	✓	
MLCatchUp [60]	✓		✓	✓	✓
APIScanner [103]		✓		✓	
Relancer [131]				✓	✓

Table 9: Detection and Repair of Dependency Conflicts

Tool	Infer TPL	Infer Python	Infer LLL
PyEGo [124]	✓	✓	✓
DockerizeMe [63]	✓		✓
SnifferDog [110]	✓		
pipreqs [52]	✓		
PyDFix [84]	✓		
V2 [64]	✓		
Watchman [111]	✓		
PyCRE [42]	✓		
smartPip [106]	✓		

API version of Numpy that TensorFlow compiled against is 0xb, not installed 0xa [18]. Upgrading the Numpy version fixed the issue.

Finding 18: Most (70.4%) of the issues caused by UC and all VMCRL-induced issues were solved by changing the component version.

5 RQ3: TOOL SURVEY

We conducted a tool survey to study the current research on automated addressing DL compatibility issues.

Tool Selection Criteria and Study Approach. We collected a total of 1,779 research papers from the top SE conferences, i.e., ICSE, ASE, and FSE, published in ACM Proceedings over the past five years (18-22). To ensure up-to-date information, we also included the accepted papers of ICSE 2023, which were available on the official website prior to our submission.

- First, we reviewed the 1,779 collected papers to identify papers that meet the following filtering criteria: (1) The paper introduces a tool specifically designed to address issues related to *Python* **AND** (*dependency conflicts* **OR** *API evolution* **OR** *compatibility issues*); (2) The tool is readily available for download and use. Through this process, we extracted nine papers.
- Next, we continued the identification process by examining the references and citations of the nine papers obtained in the first step, applying the same filtering criteria. At this stage, we identified additional five papers and one tool.
- Finally, we extracted a total of 15 tools, consisting of 14 papers and one tool from GitHub. We classified these tools into two categories, i.e., library API evolution, detection and repair of dependency conflicts, as shown in Table 8 and Table 9, respectively.

Two authors independently reviewed and identified all the papers. The inter-rater agreement was measured using Cohen's kappa coefficient, which exceeded 95% after their inspection. In cases of inconsistency, a third author participated in the discussion to reach a consensus and finalize the results. Besides, we also configured and executed these tools to evaluate their functionality.

Tool Survey Results. The tools listed in Table 8 that are related to library API evolution, can be categorized into two main approaches: static and dynamic. The static approach involves establishing a knowledge base of library API evolution histories using manual or semi-automatic techniques. The Python project source code is then parsed to extract the invoked APIs through AST. The extracted APIs are subsequently matched against the library APIs stored in the prebuilt knowledge base. For example, MLCatchUp [60], DLocator [109], PyCompat [129], and APIScanner [103] employ the static method to identify deprecated APIs or APIs with breaking changes in Python projects. Dynamic methods are used by tools like AexPy [44] and Relancer [131]. AexPy uses dynamic reflection to infer the types, parameters, aliases, and inheritance, making the API evolution knowledge base more comprehensive. Relancer adopts an iterative approach by executing code snippets in Jupyter Notebook and iteratively repairing deprecated APIs based on the runtime error messages.

Besides, tools (Table 9) for resolving dependency conflicts in Python programs can be divided into static and dynamic approaches. In the static method, the dependency relationships between libraries and their required versions are collected from platforms such as the Python-PyPI repository and Libraries.io. Tools like PyEgo [124], PyCRE [42], DockerizeMe [63], and Watchman [111] convert this collected data into a dependency knowledge graph. They then parse the project's AST to identify the resources (e.g., APIs) utilized by libraries, and use these resources as query conditions. By designing appropriate traversal algorithms, they resolve the dependencies or version conflicts of libraries in the project by traversing the knowledge graph. smartPip [106] extracts version constraints between TPLs from a prebuilt knowledge base. These constraints are then transformed into satisfiability modulo theories (SMT) expressions. By solving these SMT expressions, smartPip effectively resolves version dependencies between TPLs. pipreqs [52] and SnifferDog [110] adopt a different approach. They first parse the project's AST to extract the APIs used. They then compare these APIs with the entries in a prebuilt database to infer the corresponding TPLs used in the project. Dynamic methods are also employed by tools such as V2 [64] and PyDFix [84]. V2 builds on DockerizeMe [63] and uses runtime error messages to narrow down the search space for candidate TPLs. PyDFix tackles dependency issues in large Python projects by running the project's build process (e.g., `pip install` and `python setup.py`) and analyzing the dependency errors found in the build log.

Finding 19: Of the six tools related to library API evolution, none of them can fully realize the entire automation process from detection to repair of DL compatibility issues caused by API incompatibility. For the resolution of dependency conflicts in Python programs, nine tools attempt to fix dependency conflicts induced by *TPL-TPL*, but only a few of them can infer the incompatibilities caused by the Python interpreter versions and system libraries. Currently, none of them can detect and fix compatibility issues caused by CUDA/cuDNN which are commonly used in DL systems.

6 DISCUSSIONS AND IMPLICATIONS

6.1 Comparison with Traditional Software

Python Programs. In Python programs, compatibility issues arise primarily from API incompatibility [44, 129] and dependency conflicts between libraries [42, 124]. Since our study focuses on Python-based DL systems (as defined in Section 2), CORE-TPL and TPL-TPL are the most common types (57.7%) of compatibility issues in this domain. DL systems, characterized by their data-driven nature, demand substantial computational resources (e.g., GPUs and TPUs), to expedite model training. DL frameworks rely heavily on low-level libraries, particularly CUDA and cuDNN, to take advantage of the computational capabilities of GPUs. As a result, compatibility issues can arise between DL libraries and the underlying components, especially within DL systems. Notably, TPL-LLL, TPL-Hardware, and LLL-Hardware are common types in DL systems, accounting for 24.4% of the total compatibility issues.

Android Applications. In Android applications, compatibility issues are mainly related to variations in device/platform configurations [113, 115], evolutionary changes in the OS [80, 117], and callback compatibility [65]. Similar to our findings, compatibility issues in Android occur during both the installation and execution/runtime stages [37]. During installation, the majority of failures are due to applications using library functions that are not supported by the underlying architecture. Meanwhile, runtime crashes often stem from API incompatibility at different levels resulting from OS evolution. There are significant differences in compatibility issues between Android applications and DL systems. For Android applications, compatibility issues are primarily caused by fragmentation, where certain versions of the Android OS can only run on certain devices. On the other hand, in DL systems, incompatibility with hardware devices often occurs due to version mismatch between the DL framework and the underlying hardware (TPL-Hardware). Another difference is the nature of breaking changes. In Android, breaking changes typically result from the evolution of the OS. In contrast, DL systems exhibit greater independence at the OS level, with fewer dependencies on specific OS versions. Besides, the API changes are primarily driven by library evolution.

Web Applications. Compatibility issues in web applications encompass various aspects, including cross-browser compatibility [82], and web API evolution-related compatibility [73, 123]. The first type is specific to web applications. In addition to disruptive changes such as adding or removing methods and parameters, API evolution in web applications often involves issues like unsupported request access and API access restrictions. These issues, specific to web API evolution, have not been observed in DL systems [96].

Others. Linux system compatibility is mainly reflected in kernel version compatibility [56], application version compatibility [89, 95], and system API call compatibility [34]. The root cause of these compatibility issues often stems from the differences in the versions of these components. To maintain backward compatibility, Linux developers often choose to rename old APIs, e.g., from `vm86()` to `vm86old()` [34]. Besides, similar to DL systems, compatibility issues in Java software systems mainly include API evolution and component compatibility [69]. However, Java software systems put more emphasis on maintaining backward compatibility of components compared to our study, which considers both forward and backward compatibility. In addition, the peculiarities of the Java language, such as the explicit specification of class and method modifiers [116], result in API evolution patterns that differ from those observed in DL systems.

In summary, compatibility issues in DL systems differ from those in traditional software due to their unique characteristics and requirements. Generalizations about traditional software may not be directly applicable to DL systems.

6.2 Implications

Ensuring Consistency Between API Usage and Installed Library Versions. A large proportion of DL compatibility issues are caused by API incompatibility, especially the *CORE-TPL* and *TPL-TPL* (Findings 1, 2, and 12). DL developers should pay attention to the consistency between the used APIs and the installed library version. In particular, when encountering exceptions such as *AttributeError*, *ImportError*, and *TypeError* during execution (Finding 14), developers are suggested to consider the potential compatibility issues. For issues caused by the API invocation between TPL and TPL, a common practice is to reinstall the libraries that satisfy the version constraints. We suggest using tools like PyEGo [124] to resolve the version conflicts between TPLs (Finding 19).

Early Determination of Component Versions. Due to the data-driven nature of DL, these systems rely heavily on high-performance computing resources (e.g., GPUs) to accelerate the model training process, resulting in a high proportion of TPL-LLL, TPL-Hardware, and LLL-Hardware issues (Findings 3, 5, and 8). Such dependencies often lead to breaking impact (Findings 10 and 11). Therefore, to maintain the correct configuration of DL systems, developers should carefully examine the official documents to determine the appropriate versions of components. For example, the tested compatible versions of CUDA/cuDNN for different TensorFlow versions [102], the compatible versions between CUDA/cuDNN and drivers [87], and the GPU architectures [94].

Challenges in Automated Detection and Repair of API Evolution Issues. (1) Construct API Mappings: Two types of mappings need to be established. The first mapping is between the API calls in the project and their corresponding definitions in TPL. Current approaches mainly rely on matching the API names. For example, DLocator [109] formats all API calls into a fully qualified name format (i.e., `A.B.C.API_Name`) to represent the hierarchical structure of the entire API path. The goal is to eliminate the impact of aliases and allow for accurate mapping to the TPL API. However, there are two problems with this approach. First, the Python import mechanism can cause inconsistencies between the actual call path and the actual API path in the source code, especially for complex DL frameworks. For example, the API `torch.load` in PyTorch version 1.5.0 has the real path `torch.serialization.load` in the source code. Other APIs with the same name include `torch.hub.load` and `torch.jit.__init__.load`. This inconsistency leads to multiple match results. Second, complex DL frameworks like PyTorch, which have many built-in APIs compiled in C/C++, often have multiple overloaded APIs. For example, `torch.max` has three overloads in PyTorch version 1.5.0 [90], making it challenging to determine the correct mapping between the called API and its definition in the TPL. The second mapping involves establishing relationships between TPL APIs across different versions to analyze API evolution. MLCatchUp [60] creates this mapping manually because it is difficult to automate. Mapping becomes particularly challenging when TPL APIs are renamed or removed. (2) Repair and Verification: This phase aims to repair and verify the invoked APIs that have breaking changes. However, automatically switching to another API when repairing a specific API fails is a non-trivial task. Sequential code execution makes it difficult to continue the repair process. For example, when Relancer [131] fails to repair the first encountered

API, it cannot continue to repair other APIs with breaking changes in the code snippet.

Challenges in Automated Detection and Resolution of Incompatible Component Versions. (1) Python Interpreter Identification: Inferring the specific version of the Python interpreter is currently done using the Python standard libraries and syntax features employed in Python projects [124]. However, this approach becomes challenging when projects don't use standard libraries or when explicit syntax features are absent. (2) Low-level Library Dependency Identification: Resolving compatibility issues that arise from low-level libraries such as CUDA/cuDNN, which depend on specific GPU hardware models, poses significant difficulties and may require low-level code dependence analysis [99, 100]. (3) OS-specific Dependency Identification: Addressing compatibility issues stemming from TPL-OS interactions is a non-trivial task.

7 THREATS TO VALIDITY

Internal Threat. The main internal threat comes from the subjective bias and error in the manual classification and labeling of DL compatibility issues. We followed the open coding method in classification and constantly optimized the classification according to the iterative rounds. To minimize subjective bias, two authors independently classified the posts, compared their results, and resolved any inconsistencies through discussion with a third author until a consensus was reached. Besides, we tried to reproduce the reported issues to facilitate the classification process. Moreover, the dataset is publicly available for further investigation and replication.

External Threat. Our dataset is sourced exclusively from the SO, without incorporating data from other sources. This poses a potential threat to the generalizability of our findings. Besides, our study relies on the selection of 12 keywords to filter the data. However, this approach may inadvertently exclude relevant posts, thus introducing a potential bias. To alleviate this threat, we selected six common keywords related to Python program exceptions, which covered about 54.4% of the 3,072 posts. Moreover, the inclusion of posts related to Theano, a discontinued DL library, may limit the applicability of our results. However, there are only 16 compatibility posts tagged with Theano, which is a relatively small proportion.

8 RELATED WORK

Empirical Study on DL Bugs. The characteristics of various DL bugs have been widely studied, e.g., DL bugs taxonomy [39, 45, 67, 68, 86, 108, 127], cloud API misuse [105], DL job failures [125], performance issues [38, 81], development and deployment faults [40, 41, 126], DL libraries and compiler bugs [46, 70, 79, 93], and model optimization bugs [57]. For example, Yang *et al.* [122] conducted an analysis of DL bugs on Github and found that certain bugs were the result of API evolution and incorrect configurations. Zhang *et al.* [125] highlighted that API in DL frameworks can lead to failures in DL jobs. However, these studies did not specifically focus on compatibility issues. Our paper comprehensively analyzes the characteristics of DL compatibility issues.

Analysis of API Evolution in Python Libraries. Many studies focus on the analyzing API evolution [91, 128, 129], detecting and evaluating API changes [35, 44, 60, 103, 109] in Python projects. Zhang *et al.* [129] conducted the first large-scale and fine-grained study of the evolution patterns of Python libraries. Du *et al.* [44] proposed AexPy, an API-model-based approach that outperforms existing tools in detecting breaking changes in Python libraries.

Repairing Dependency Conflicts in Python Programs. Various studies have focused on automating the detection and resolution of dependency conflicts between TPLs in Python projects [42, 63, 64, 84, 106, 110, 111, 124]. The state-of-the-art, PyEgo, proposed by Ye *et al.* [124], automatically resolves compatibility issues related to the Python interpreter, TPLs, and system libraries.

Compatibility Issues in other Software Systems. Compatibility issues have been widely studied in Android apps [37, 65, 66, 78, 80, 92, 97, 101, 113, 114, 117]. Many studies focus on detecting deprecated APIs [61, 74, 76, 77, 120], and developing tools to resolve compatibility issues [48, 58, 59, 72, 75, 115]. For example, Zhao *et al.* [130] proposed RepairDroid, a tool designed to address three types of compatibility issues in Android apps: OS-induced, device-specific, and inter-callback compatibility issues. Besides, the continuous evolution of web APIs and client apps often leads to compatibility issues related to API evolution and cross-browser behavior [73, 82, 96, 123]. Furthermore, several studies have been conducted to investigate compatibility issues in other domains, such as Linux [34, 56, 89, 95], Java [69, 116], C/C++ [71], and JavaScript [85]. Our study presents the first comprehensive empirical study of compatibility issues in DL systems, distinguishing it from previous research on traditional software.

9 CONCLUSION

In this paper, we conducted the first comprehensive empirical study to characterize compatibility issues in DL systems. We identified 352 DL compatibility issues from 3,072 Stack Overflow posts. By analyzing these issues, we proposed a taxonomy of DL compatibility issues and learned their manifestation stages and symptoms. We further summarized three root causes and two common fixing strategies. Moreover, we conducted a tool survey to investigate the current research state on automated detection and repair tools for DL compatibility issues. We believe that this study can provide researchers and practitioners with a better understanding of DL compatibility issues and facilitate future research in related areas. In the future, we plan to leverage the findings to develop tools for detecting and repairing DL compatibility issues.

10 DATA AVAILABILITY

The replication package and the dataset are publicly available at https://doi.org/10.5281/zenodo.8207011.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by National Natural Science Foundation of China under Grant 62002163, Natural Science Foundation of Jiangsu Province under Grant BK20200441, Australian Research Council under Grants DP210101348 and FT220100391, and the National Key Research and Development Program of China under Grant 2019YFE0198100. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

REFERENCES

[1] StackOverflow.com 32444016. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/32444016.
[2] StackOverflow.com 33651810. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/33651810.
[3] StackOverflow.com 33671372. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/33671372.
[4] StackOverflow.com 38546672. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/38546672.
[5] StackOverflow.com 41005249. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/41005249.
[6] StackOverflow.com 42456461. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/42456461.
[7] StackOverflow.com 42675391. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/42675391.
[8] StackOverflow.com 42823627. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/42823627.
[9] StackOverflow.com 43069519. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/43069519.
[10] StackOverflow.com 44993098. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/44993098.
[11] StackOverflow.com 48383846. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/48383846.
[12] StackOverflow.com 48929098. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/48929098.
[13] StackOverflow.com 49079990. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/49079990.
[14] StackOverflow.com 51320027. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/51320027.
[15] StackOverflow.com 52906186. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/52906186.
[16] StackOverflow.com 53467011. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/53467011.
[17] StackOverflow.com 53707068. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/53707068.
[18] StackOverflow.com 53765453. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/53765453.
[19] StackOverflow.com 53950186. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/53950186.
[20] StackOverflow.com 55261785. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/55261785.
[21] StackOverflow.com 56406862. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/56406862.
[22] StackOverflow.com 57122907. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/57122907.
[23] StackOverflow.com 57681910. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/57681910.
[24] StackOverflow.com 57718512. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/57718512.
[25] StackOverflow.com 58901682. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/58901682.
[26] StackOverflow.com 59226533. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/59226533.
[27] StackOverflow.com 59493606. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/59493606.
[28] StackOverflow.com 59894720. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/59894720.
[29] StackOverflow.com 65988678. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/65988678.
[30] StackOverflow.com 69865825. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/69865825.
[31] StackOverflow.com 71174306. 2023. Retrieved January 10, 2023 from https://stackoverflow.com/questions/71174306.
[32] Anaconda. 2023. Retrieved January 10, 2023 from https://www.anaconda.com/.
[33] archive.org. 2023. stackexchange. Retrieved January 10, 2023 from https://archive.org/details/stackexchange_20221005.
[34] Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E Hassan, Juergen Dingel, and James R Cordy. 2018. Analyzing a Decade of Linux System Calls. *EMSE* (2018).
[35] Wilson Baker, Michael O'Connor, Seyed Reza Shahamiri, and Valerio Terragni. 2022. Detect, Fix, and Verify TensorFlow API Misuses. In *SANER*.
[36] bazel.com. 2023. Bazel. Retrieved January 10, 2023 from https://bazel.build/.
[37] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A Large-scale Study of Application Incompatibilities in Android. In *ISSTA*.
[38] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding Performance Problems in Deep Learning Systems. In *FSE*.
[39] Junjie Chen, Yihua Liang, Qingchao Shen, and Jiajun Jiang. 2022. Toward Understanding Deep Learning Framework Bugs. *TOSEM* (2022).
[40] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A Comprehensive Study on Challenges in Deploying Deep Learning Based Software. In *FSE*.
[41] Zhenpeng Chen, Huihan Yao, Yiling Lou, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, and et al. 2021. An Empirical Study on Deployment Faults of Deep Learning Based Mobile Applications. In *ICSE*.
[42] Wei Cheng, Xiangrong Zhu, and Wei Hu. 2022. Conflict-aware Inference of Python Compatible Runtime Environments with Domain Knowledge Graph. In *ICSE*.
[43] Danny Dig and Ralph Johnson. 2006. How do APIs Evolve? A Story of Refactoring. *SMR* (2006).
[44] Xingliang Du and Jun Ma. 2022. AexPy: Detecting API Breaking Changes in Python Packages. In *ISSRE*.
[45] Xiaoting Du, Yulei Sui, Zhihao Liu, and Jun Ai. 2022. An Empirical Study of Fault Triggers in Deep Learning Frameworks. *TDSC* (2022).
[46] Xiaoting Du, Guanping Xiao, and Yulei Sui. 2020. Fault Triggers in the TensorFlow Framework: An Experience Report. In *ISSRE*.
[47] Xiaoting Du, Zheng Zheng, Guanping Xiao, Zenghui Zhou, and Kishor S Trivedi. 2021. DeepSIM: Deep Semantic Information-based Automatic Mandelbug Classification. *TRel* (2021).
[48] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage Update for Android Apps. In *ISSTA*.
[49] GitHub.com. 2023. commit 76f7c02 of tensorflow. Retrieved January 10, 2023 from https://github.com/tensorflow/tensorflow/commit/333dc32ff79af21484695157f3d141dc776f7c02.
[50] GitHub.com. 2023. commit 9a1b9ef of keras. Retrieved January 10, 2023 from https://github.com/keras-team/keras/commit/d663fda862df1c831e7f93f1e3feb2e189a1b9ef.
[51] GitHub.com. 2023. commit c33da89 of tensorflow. Retrieved January 10, 2023 from https://github.com/tensorflow/tensorflow/commit/7a8c63da365106048dc96affddb39e2fdc33da89.
[52] Github.com. 2023. pipreqs. Retrieved January 10, 2023 from https://github.com/bndr/pipreqs.
[53] Github.com. 2023. Tensorflow 1.6.0. Retrieved January 10, 2023 from https://github.com/tensorflow/tensorflow/releases/tag/v1.6.0.
[54] GitHub.com. 2023. TensorFlow 2.0.0. Retrieved January 10, 2023 from https://github.com/tensorflow/tensorflow/releases/tag/v2.0.0.
[55] GitHub.com. 2023. TensorFlow 2.4.0. Retrieved January 10, 2023 from https://github.com/keras-team/keras/releases/tag/2.4.0.
[56] Michael W. Godfrey and Qiang Tu. 2000. Evolution in Open Source Software: A Case Study. In *ICSM*.
[57] Hao Guan, Ying Xiao, Lijia Ying, Yepang Liu, and Guangdong Bai. 2023. A Comprehensive Study of Real-world Bugs in Machine Learning Model Optimization. In *ICSE*.
[58] Stefanus A Haryono, Ferdian Thung, Hong Jin Kang, Lucas Serrano, Gilles Muller, Julia Lawall, and et al. 2020. Automatic Android Deprecated-API Usage Update by Learning from Single Updated Example. In *ICPC*.
[59] Stefanus A Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, Hong Jin Kang, and et al. 2022. AndroEvolve: Automated Android API Update With Data Flow Analysis and Variable Denormalization. *EMSE* (2022).
[60] Stefanus A Haryono, Ferdian Thung, David Lo, Julia Lawall, and Lingxiao Jiang. 2021. MLCatchUp: Automated Update of Deprecated Machine-learning APIs in Python. In *ICSME*.
[61] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. 2018. Understanding and Detecting Evolution-induced Compatibility Issues in Android Apps. In *ASE*.
[62] Mohammad Hesam Hesamian, Wenjing Jia, Xiangjian He, and Paul Kennedy. 2019. Deep Learning Techniques for Medical Image Segmentation: Achievements and Challenges. *JDI* (2019).
[63] Eric Horton and Chris Parnin. 2019. Dockerizeme: Automatic Inference of Environment Dependencies for Python Code Snippets. In *ICSE*.
[64] Eric Horton and Chris Parnin. 2019. V2: Fast Detection of Configuration Drift in Python. In *ASE*.
[65] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and Detecting Callback Compatibility Issues for Android Applications. In *ASE*.
[66] Huaxun Huang, Ming Wen, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing and Detecting Configuration Compatibility Issues in Android Apps. In *ASE*.
[67] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *ICSE*.
[68] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *FSE*.
[69] Kamil Jezek and Jens Dietrich. 2017. API Evolution and Compatibility: A Data Corpus and Tool Evaluation. *J. Object Technol.* (2017).

[70] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An Empirical Study on Bugs Inside TensorFlow. In *DASFAA*.

[71] Zhouyang Jia, Shanshan Li, Tingting Yu, Chen Zeng, Erci Xu, Xiaodong Liu, and et al. 2021. DepOwl: Detecting Dependency Bugs to Prevent Compatibility Failures. In *ICSE*.

[72] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2020. A3: Assisting Android API Migrations Using Code Examples. *TSE* (2020).

[73] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. 2013. How Does Web Service API Evolution Affect Clients?. In *ICWS*.

[74] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. 2016. Accessing Inaccessible Android APIs: An Empirical Study. In *ICSME*.

[75] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the Detection of API-related Compatibility Issues in Android Apps. In *ISSTA*.

[76] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2018. Characterising Deprecated Android APIs. In *MSR*.

[77] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2020. Cda: Characterising Deprecated Android APIs. *EMSE* (2020).

[78] Pei Liu, Yanjie Zhao, Haipeng Cai, Mattia Fazzini, John Grundy, and Li Li. 2022. Automatically Detecting API-induced Compatibility Issues in Android Apps: A Comparative Analysis (Replicability Study). In *ISSTA*.

[79] Zhihao Liu, Yang Zheng, Xiaoting Du, Zheng Hu, Wenjie Ding, Yanming Miao, and et al. 2022. Taxonomy of Aging-related Bugs in Deep Learning Libraries. In *ISSRE*.

[80] Tarek Mahmud, Meiru Che, and Guowei Yang. 2021. Android Compatibility Issue Detection Using API Differences. In *SANER*. 480–490.

[81] Tarek Makkouk, Dong Jae Kim, and Tse-Hsun Peter Chen. 2022. An Empirical Study on Performance Bugs in Deep Learning Frameworks. In *ICSME*.

[82] Ali Mesbah and Mukul R Prasad. 2011. Automated Cross-browser Compatibility Testing. In *ICSE*.

[83] Sajjad Mozaffari, Omar Y Al-Jarrah, Mehrdad Dianati, Paul Jennings, and Alexandros Mouzakitis. 2020. Deep Learning-based Vehicle Behavior Prediction for Autonomous Driving Applications: A Review. *TITS* (2020).

[84] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing Dependency Errors for Python Build Reproducibility. In *ISSTA*.

[85] Romulo Nascimento, Andre Hora, and Eduardo Figueiredo. 2022. Exploring API Deprecation Evolution in JavaScript. In *SANER*.

[86] Amin Nikanjam, Mohammad Mehdi Morovati, Foutse Khomh, and Houssem Ben Braiek. 2022. Faults in Deep Reinforcement Learning Programs: A Taxonomy and A Detection Approach. *ASE J* (2022).

[87] Nvidia.com. 2023. CUDA Compatibility. https://docs.nvidia.com/deploy/cuda-compatibility/index.html.

[88] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, and et al. 2020. Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance. In *ASE*.

[89] Andrey Ponomarenko and Vladimir Rubanov. 2011. Automatic Backward Compatibility Analysis of Software Component Binary Interfaces. In *CSAE*.

[90] pytorch.org. 2023. torch.max. Retrieved June 14, 2023 from https://pytorch.org/docs/1.5.0/torch.html#torch.max.

[91] Haowei Quan, Jiawei Wang, Bo Li, Xiaoning Du, Kui Liu, and Li Li. 2022. Characterizing Python Method Evolution with PyMevol: An Essential Step Towards Enabling Reliable Software Systems. In *ISSREW*.

[92] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. 2019. Data-driven Solutions to Detect API Compatibility Issues in Android: An Empirical Study. In *MSR*.

[93] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A Comprehensive Study of Deep Learning Compiler Bugs. In *FSE*.

[94] Arnon Shimoni. 2023. Retrieved June 14, 2023 from https://arnon.dk/matching-sm-architectures-arch-and-gencode-for-various-nvidia-cards/.

[95] Pavel Shved and Denis Silakov. 2009. Binary Compatibility of Shared Libraries Implemented in C++ on GNU/Linux Systems. In *SYRCoSE*.

[96] SM Sohan, Craig Anslow, and Frank Maurer. 2015. A Case Study of Web API Evolution. In *SERVICES*.

[97] Zihe Song, Yingfeng Chen, Lei Ma, Shangjie Lu, Honglei Lin, Changjie Fan, and et al. 2022. An Empirical Analysis of Compatibility Issues for Industrial Mobile Games (Practical Experience Report). In *ISSRE*.

[98] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2vec: Value-flow-based Precise Code Embedding. OOPSLA (2020).

[99] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-flow Analysis in LLVM. In *CC*.

[100] Yulei Sui and Jingling Xue. 2018. Value-flow-based Demand-driven Pointer Analysis for C and C++. *TSE* (2018).

[101] Xiaoyu Sun, Xiao Chen, Yanjie Zhao, Pei Liu, John Grundy, and Li Li. 2022. Mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues. In *ASE*.

[102] tensorflow.org. 2023. TensorFlow build guide. Retrieved January 10, 2023 from https://www.tensorflow.org/install/source#tested_build_configurations.

[103] Aparna Vadlamani, Rishitha Kalicheti, and Sridhar Chimalakonda. 2021. APIScanner-Towards Automated Detection of Deprecated APIs in Python Libraries. In *ICSE-Companion*.

[104] Anthony J Viera, Joanne M Garrett, et al. 2005. Understanding Interobserver Agreement: The Kappa Statistic. *Fam med* (2005).

[105] Chengcheng Wan, Shicheng Liu, Henry Hoffmann, Michael Maire, and Shan Lu. 2021. Are Machine Learning Cloud APIs Used Correctly?. In *ICSE*.

[106] Chao Wang, Rongxin Wu, Haohao Song, Jiwu Shu, and Guoqing Li. 2022. smartPip: A Smart Approach to Resolving Python Dependency Conflict Issues. In *ASE*.

[107] Dinghua Wang, Shuqing Li, Guanping Xiao, Yepang Liu, and Yulei Sui. 2021. An Exploratory Study of Autopilot Software Bugs in Unmanned Aerial Vehicles. In *FSE*.

[108] Gan Wang, Zan Wang, Junjie Chen, Xiang Chen, and Ming Yan. 2022. An Empirical Study on Numerical Bugs in Deep Learning Programs. In *ASE*.

[109] Jiawei Wang, Li Li, Kui Liu, and Haipeng Cai. 2020. Exploring How Deprecated Python Library APIs Are (Not) Handled. In *FSE*.

[110] Jiawei Wang, Li Li, and Andreas Zeller. 2021. Restoring Execution Environments of Jupyter Notebooks. In *ICSE*.

[111] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, and et al. 2020. Watchman: Monitoring Dependency Conflicts For Python Library Ecosystem. In *ICSE*.

[112] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2022. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. *TOSEM* (2022).

[113] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *ASE*.

[114] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2019. Pivot: Learning API-device Correlations to Facilitate Android Compatibility Issue Detection. In *ICSE*.

[115] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. 2018. Understanding and Detecting Fragmentation-induced Compatibility Issues for Android Apps. *TSE* (2018).

[116] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and Impact Analysis of API Breaking Changes: A Large-scale Study. In *SANER*.

[117] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, and et al. 2020. How Android Developers Handle Evolution-induced API Compatibility Issues: A Large-scale Study. In *ICSE*.

[118] Guanping Xiao, Xiaoting Du, Yulei Sui, and Tao Yue. 2020. Hindbr: Heterogeneous Information Network based Duplicate Bug Report Prediction. In *ISSRE*.

[119] Guanping Xiao, Jun Liu, Zheng Zheng, and Yulei Sui. 2021. Nondeterministic Impact of CPU Multithreading on Training Deep Learning Systems. In *ISSRE*.

[120] Guowei Yang, Jeffrey Jones, Austin Moninger, and Meiru Che. 2018. How Do Android Operating System Updates Impact Apps?. In *MOBILESoft*.

[121] Xin-Li Yang, David Lo, Xin Xia, Zhi-Yuan Wan, and Jian-Ling Sun. 2016. What Security Questions Do Developers Ask? A Large-scale Study of Stack Overflow Posts. *JCST* (2016).

[122] Yilin Yang, Tianxing He, Zhilong Xia, and Yang Feng. 2022. A Comprehensive Empirical Study on Bug Characteristics of Deep Learning Frameworks. *IST* (2022).

[123] Jerin Yasmin, Yuan Tian, and Jinqiu Yang. 2020. A First Look at The Deprecation of RESTful APIs: An Empirical Study. In *ICSME*.

[124] Hongjie Ye, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. 2022. Knowledge-Based Environment Dependency Inference for Python Programs. In *ICSE*.

[125] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An Empirical Study on Program Failures of Deep Learning Jobs. In *ICSE*.

[126] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *ISSRE*.

[127] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *ISSTA*.

[128] Zejun Zhang, Yanming Yang, Xin Xia, David Lo, Xiaoxue Ren, and John Grundy. 2021. Unveiling the Mystery of API Evolution in Deep Learning Frameworks: A Case Study of TensorFlow 2. In *ICSE-SEIP*.

[129] Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. 2020. How Do Python Framework APIs Evolve? An Exploratory Study. In *SANER*.

[130] Yanjie Zhao, Li Li, Kui Liu, and John Grundy. 2022. Towards Automatically Repairing Compatibility Issues in Published Android Apps. In *ICSE*.

[131] Chenguang Zhu, Ripon K Saha, Mukul R Prasad, and Sarfraz Khurshid. 2021. Restoring the Executability of Jupyter Notebooks by Automatic Upgrade of Deprecated APIs. In *ASE*.

[132] Jianfei Zhu, Guanping Xiao, Zheng Zheng, and Yulei Sui. 2022. Enhancing Traceability Link Recovery with Unlabeled Data. In *ISSRE*.

Received 2023-02-02; accepted 2023-07-27