Unraveling Code Clone Dynamics in Deep Learning Frameworks

MARAM ASSI, Université du Québec à Montréal, Canada
SAFWAT HASSAN, University of Toronto, Canada
YING ZOU, Queen's University, Canada

Deep Learning (DL) frameworks play a critical role in advancing artificial intelligence, and their rapid growth underscores the need for a comprehensive understanding of software quality and maintainability. DL frameworks, like other systems, are prone to code clones. Code clones refer to identical or highly similar source code fragments within the same project or even across different projects. Code cloning can have positive and negative implications for software development, influencing maintenance, readability, and bug propagation. While the existing studies focus on studying clones in DL-based applications, to our knowledge, no work has been done investigating clones, their evolution and their impact on the maintenance of DL frameworks. In this paper, we aim to address the knowledge gap concerning the evolutionary dimension of code clones in DL frameworks and the extent of code reuse across these frameworks. We empirically analyze code clones in nine popular DL frameworks, i.e., *TensorFlow*, *Paddle*, *PyTorch*, *Aesara*, *Ray*, *MXNet*, *Keras*, *Jax* and *BentoML*, to investigate (1) the characteristics of the long-term code cloning evolution over releases in each framework, (2) the short-term, i.e., within-release, code cloning patterns and their influence on the long-term trends, and (3) the file-level code clones within the DL frameworks. Our findings reveal that DL frameworks adopt four distinct cloning trends: *"Serpentine"*, *"Rise and Fall"*, *"Decreasing"*, and *"Stable"* and that these trends present some common and distinct characteristics. For instance, bug-fixing activities persistently happen in clones irrespective of the clone evolutionary trend but occur more in the *"Serpentine"* trend. Moreover, the within-release level investigation demonstrates that short-term code cloning practices impact long-term cloning trends. The cross-framework code clone investigation reveals the presence of *functional* and *architectural adaptation* file-level cross-framework code clones across the nine studied frameworks. We provide insights that foster robust clone practices and collaborative maintenance in the development of DL frameworks.

CCS Concepts: • **Software and its engineering → Software creation and management Software**; • **Maintaining software**;

Additional Key Words and Phrases: deep learning frameworks, code clones, clone genealogy

Authors' addresses: Maram Assi, assi.maram@uqam.ca, Université du Québec à Montréal, Montréal, Québec, Canada; Safwat Hassan, safwat.hassan@utoronto.ca, University of Toronto, Toronto, Ontario, Canada; Ying Zou, ying.zou@queensu.ca, Queen's University, Kingston, Ontario, Canada.

ACM 1049-331X/2023/2-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Maram Assi, Safwat Hassan, and Ying Zou. 2023. Unraveling Code Clone Dynamics in Deep Learning Frameworks. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (February 2023), 50 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Deep Learning (DL) frameworks are pivotal in shaping the future of artificial intelligence and machine learning. The Global DL Market is anticipated to grow a Compound Annual Growth Rate (CAGR) of 51.1% from 2022 to 2030[1]. Moreover, empirical data highlights the substantial popularity of DL repositories on GitHub, providing further evidence of the significance of DL frameworks. For instance, the *TensorFlow* framework has attracted remarkable attention, accumulating over 150,000 stars within an eight-year span [1]. Additionally, *TensorFlow* attains an impressive count of over 88,000 forks, making it among the top 5 globally recognized repositories on GitHub. These metrics affirm the widespread acknowledgment of the pivotal role of these frameworks in the artificial intelligence software development landscape.

Code cloning is the replication or duplication of source code fragments within a software project [67]. It can manifest as either exact or similar copies of code segments. While code cloning might alleviate developers' workload [34], multiple studies have demonstrated the potential negative impact of code clones on the development and maintenance of software systems [38, 43]. For instance, Mondal et al. [46] find that cloned code is associated with higher maintenance costs, especially with Type 2 and Type 3 clones in traditional systems. Moreover, domain-specific systems face unique challenges. Higo et al. [20] demonstrate that web-based systems with a high prevalence of code clones pose significant testing challenges. Similarly, Islam et al. [26] find that small clone fragments (i.e., 1 to 4 lines) are significantly more bug-prone than larger clones. Therefore, developers should be aware of the cost of the risk of code cloning. The evolution and impact of code clones are well studied in traditional systems [6, 50, 63, 76].

DL code differs from traditional system code in terms of the development paradigm and coding practices [2]. A recent study demonstrates that clone occurrences are higher in DL systems as compared to traditional code [30]. Jebnoun et al. [30] find that 9.28% of DL-related cloned code is linked to model evaluation, with 89% of those clones relating to performance metric computations. As DL models grow in complexity and scale, frequently duplicated metric y computations can lead to inefficiencies in resource usage resulting in increased memory usage and execution time, directly affecting performance and scalability. However, there is limited research to investigate if the clone analysis of traditional software could be adapted to DL software. Despite the rising popularity of DL software, only two prior studies have explored code clones within the DL domain. Jebnoun et al. [30] study clones in Python, C#, and Java-based DL applications, shedding light on the prevalence of code clones during model creation, training, and data preprocessing. In a recent study, Mo et al. [46]

[1]https://www.acumenresearchandconsulting.com/deep-learning-market

emphasize co-changed clones within DL applications. While the existing research focuses on studying clones in DL-based applications, to our knowledge, no work has investigated clones, their evolution and their impact on the maintenance of DL frameworks. Understanding the evolution of code clones in DL frameworks provides valuable insights into how technical debt accumulates over time and how these clones impact long-term maintainability.

To address the knowledge gap concerning the evolution of code cloning and its implications within the DL frameworks, in this work, we conduct an empirical study to gain a better understanding of the evolutionary dimension of code clones in DL frameworks and the extent of code reuse across these frameworks. More specifically, we analyze long-term code cloning trends over releases and short-term code cloning patterns within releases. Our goal is to offer insights on better managing and mitigating clones in DL frameworks by identifying the characteristics of short-term patterns and their impact on long-term trends. Additionally, we conduct a cross-framework clone study to provide insights into the code commonalities and collaborative work across DL frameworks.

We conduct experiments on nine popular DL frameworks, i.e., *TensorFlow*[2], *Paddle*[3], *PyTorch*[4], *Aesara*[5], *Ray*[6], *MXNet*[7], *Keras*[8], *Jax*[9] and *BentoML*[10]. Our empirical investigation yielded the following findings answering the studied research questions:

RQ1: What are the characteristics of the long-term trends observed in the evolution of code clones within DL frameworks over releases?
Our goal is to explore the characteristics of the long-term trends of the evolution of code clones over releases within DL frameworks. Our analysis identified four distinct long-term code clone evolution trends: *"Serpentine"*, *"Rise and Fall"*, *"Decreasing"* and *"Stable"*. These trends exhibit unique characteristics, with *"Rise and Fall"* and *"Decreasing"* trends often associated with a decline in overall clone coverage due to factors like code refactoring, third-party library usage, and feature elimination. Furthermore, our findings indicate that bug-fixing activities are consistent across all trends, but frameworks with the *"Serpentine"* trend, such as Paddle and MXNet, show a higher bug-proneness, indicated by a greater percentage of bug-fixing commits. *"Thick"* clones are more prone to bugs than *"thin"* clones across all trends. These findings emphasize the importance of optimizing code complexity, enhancing readability and maintainability through modularization and concise structures, and prioritizing thorough testing for

[2]https://github.com/tensorflow/tensorflow.git
[3]https://github.com/PaddlePaddle/Paddle.git
[4]https://github.com/pytorch/pytorch.git
[5]https://github.com/aesara-devs/aesara.git
[6]https://github.com/ray-project/ray.git
[7]https://github.com/apache/mxnet.git
[8]https://github.com/keras-team/keras.git
[9]https://github.com/google/jax.git
[10]https://github.com/bentoml/BentoML.git

complex, bug-prone clones to maintain software quality and reliability.

RQ2: What are the characteristics of within-release code cloning patterns and do these patterns contribute to the overarching long-term trends in code cloning?
Our goal is to identify the characteristics of short-term code cloning patterns within-release and explore their potential impact on the long-term trends in code cloning. Our analysis identified three within-release code cloning patterns, i.e., "Ascending", "Descending", and "Steady" patterns. We find that the withing-release patterns impact long-term code cloning trends. Our findings suggest that an "Ascending" code cloning pattern is characterized by a decreased committer involvement in clone pairs. Thus, minimizing the number of developers involved in modifying a clone pair may contribute to more consistent and well-maintained code clones. These insights underscore the need to encourage broader community engagement in clone-related activities and to explore strategies for code clone management and team collaboration.

RQ3: How do code clones manifest and evolve across different DL frameworks?
We aim to conduct a cross-framework clone detection to identify and analyze similarities in code across different DL frameworks. We find that cross-framework file-level code clones exist within DL frameworks, and they fall into two categories: functional and architectural adaptation code clones. This suggests an opportunity to establish clear guidelines and best practices for common DL functionalities as well as initiating community-driven efforts to develop and promote standardized code practices within the DL ecosystem.

The main contributions of our work are as follows:

(1) We analyze the characteristics of the evolution of code clones in DL frameworks throughout their entire lifespan. The analysis of distinct cloning trends provides insights to enhance the efficiency and maintainability of DL frameworks over time.
(2) We offer insights into code cloning development patterns within releases that influence code clone evolution. We identify factors affecting cloned code size and projecting long-term clone trends and provide insights into well-maintained frameworks within the DL framework community.
(3) We conduct a cross-framework code clone analysis within DL frameworks. Our investigation of the code commonality and clones across the frameworks can potentially motivate collaborative efforts within the DL community.
(4) We provide a replication package[11] for our approach, including the list of DL frameworks, the genealogy results obtained, and the scripts necessary to reproduce our study.

[11]https://github.com/mia1q/code-clone-DL-frameworks/

Paper organization. The remaining part of the paper is organized as follows. Section 2 presents the experimental setup. Section 3 illustrates the motivation, approach, and findings of our research questions. Section 4 discusses the implications of our work. Section 5 describes the potential threats of this study. Section 6 discusses the related work. Finally, Section 7 concludes the study and discusses future work.

2 Experiment Setup

In this section, we present the experimental setup. Figure 1 depicts the overview of our approach for analyzing code cloning in DL frameworks. In the first step, we select the DL frameworks. Then, we collect the commit and release data to construct the source code version history for each framework. Next, we extract the code clones within individual frameworks to build the clone genealogies and collect relevant metrics. In the final step, we extract the code clones across all the DL frameworks to answer RQ3.

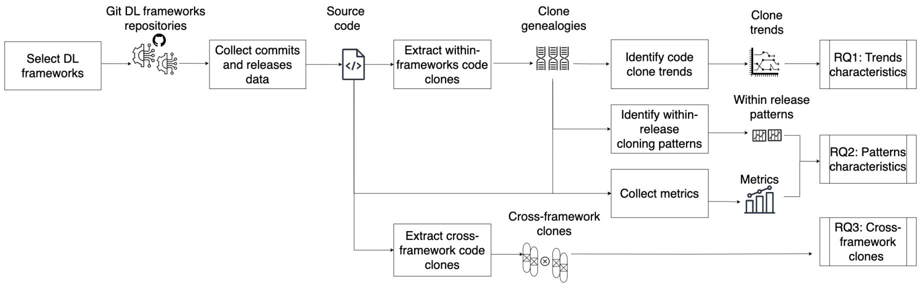
Select DL frameworks
Git DL frameworks repositories
Collect commits and releases data
Source code
Extract within-frameworks code clones
Clone genealogies
Identify code clone trends
Clone trends
RQ1: Trends characteristics
Identify within-release cloning patterns
Within release patterns
RQ2: Patterns characteristics
Collect metrics
Metrics
Extract cross-framework code clones
Cross-framework clones
RQ3: Cross-framework clones

Fig. 1. An overview of our approach for analyzing code cloning in DL frameworks.

2.1 Deep Learning frameworks selection

DL frameworks provide a robust foundation for implementing DL applications, driving significant technological advancements such as imaging and autonomous vehicles. To identify the target DL frameworks for our study, we begin by following the initial selection process outlined by Du et al. [13], which focuses on widely studied frameworks in recent literature [9, 22, 26, 30, 36, 37, 71, 79, 80]. We initially identify 14 candidate frameworks, including TensorFlow, Chainer, and Keras. These frameworks are written in Python and Java. Since Python has emerged as the most widely adopted programming language for DL applications [7], we narrow our focus to Python frameworks. After this filtering step, we end up with 9 frameworks. To ensure that the selected frameworks are indeed DL-specific, we manually verify each framework by examining their GitHub repositories, reading their README files, and confirming that they are designed for DL purposes. We end up with the following nine DL frameworks: *TensorFlow*, *Paddle*, *PyTorch*, *Aesara*, *Ray*, *MXNet*, *Keras*, *Jax* and *BentoML*.

2.2 Commit and releases data collection

We extract the historical data of each repository to study the evolution of code clones and identify the evolution trends. In particular, we collect the commit history to extract the corresponding source code over time and the release information to be able to observe the trends in code cloning across different releases.

Commit collection and source code extraction. For every framework, we obtain all commits by utilizing the `git log - pretty=format:"%h,%ae, %ai, %s"` command, which includes commit information, such as the commit log message, author and committer details, and commit dates. For every commit, we compute the snapshot size in terms of Python source lines of code (SLOC) using the Cloc library [12] version *v-1.96*.

Release collection and pre-processing. We gather the framework releases through the GitHub API, including their version numbers, release dates, and the associated release notes. A version string is composed of three numerical components delimited by periods, e.g., "1.7.2". We follow the existing work [83] and adopt semantic versioning (i.e., version strings represented by three numbers separated by dots) [39] to categorize the releases in our study into three main types: *major*, *minor*, and *patch*. *Major* releases are associated with software's API changes, *minor* releases to the addition of new features and *patch* to backward-compatible bug fixing. A major release corresponds to a modification in the first number, a minor release to a modification in the second number, and a patch release occurs in the third number. To determine the release type, we compare the version strings of the current release R_i and the preceding release R_{i-1}. If there are changes in multiple numbers, the precedence follows major over minor or patch, and minor over patch. In our study, we exclude *patch* releases and include only *major*, *minor* as they represent stable releases. Table 4 summarizes the statistics of the collected DL frameworks.

2.3 Within-framework code clone extraction

To identify code clones within one framework, we employ NiCad tool [68], a well-known and most-used text-based clone detection tool [85], which exhibits strong capabilities to detect both exact and near-miss clones at the block and function level with high precision and recall [69]. NiCad employs a hybrid approach that combines syntactic and semantic analysis for code clone detection. It primarily relies on token-based similarity measures to identify potential clones. NiCad operates through a multi-step process. It starts by normalizing the source code by stripping away formatting and comments. It then applies a sequence of transformations to the code to detect potential clones. NiCad segments the code into "functions" or code blocks, which are the units of comparison. Once the code has been normalized and segmented, NiCad uses text-based similarity measures to identify similar code blocks and outputs the detected clones. It employs customizable thresholds to classify code as clones based on their similarity. NiCad requires the source code of the project or framework to be analyzed as input. The output of NiCad is a list of clone pairs, along with their

[12]https://github.com/AlDanial/cloc

similarity scores. We use the latest available version *NiCad6.2* of NiCad [13]. We follow the standard configuration of NiCad, i.e., dissimilarity threshold of 30% and minimum cloned code size of 5 lines, used in recent related work [30, 46] as with these settings, NiCad is reported to be very accurate in clone detection. We detect three code clone types: Type 1, Type 2, or Type 3, and we choose clones at the functions level. Table 1, Table 2 and Table 3 illustrates samples of the clone types. To perform an analysis of the distribution and evolution of clone types in DL frameworks, we detect three types of code clones (Type 1, Type 2, and Type 3) . Type 1 clones are exact copies of code fragments without any modifications except for variations in whitespace and comments. Type 2 clones are syntactically identical fragments except for variations in identifiers, literals, types, layout, and comments. Type 3 clones have copied fragments with a few modifications (i.e., added, modified, or deleted statements). By default, Type 2 clones include Type 1 clones, and Type 3 clones include both Type 1 and Type 2 clones. To analyze clone types independently, we exclude Type 1 clones from Type 2 clones. We apply the same approach to exclude Type 1 and Type 2 clones from Type 3. We choose clones at the function level for all clone types detection.

```
def connection_pattern(self, node):
    # Specify that epsilon and
     running_average_factor are not
     connected to outputs.
    patterns = [
        [True, True, True],  # x
        [True, True, True],  # scale
        [True, True, True],  # bias
        [False, False, False],  #
     epsilon
        [False, False, False],  #
     running_average_factor
    ]
    # Optional running_mean and
     running_var are only
    # connected to their new values.
    for i in range(5, len(node.inputs)):
        patterns[0].append(True)
        for pattern in patterns[1:]:
            pattern.append(False)
        patterns.append([False] * (3 + i
      - 5) + [True])
    return patterns
```

```
def connection_pattern(self, node):
    # Specify that epsilon and
     running_average_factor are not
     connected to outputs.
    patterns = [
        [True, True, True],  # x
        [True, True, True],  # scale
        [True, True, True],  # bias
        [False, False, False],  #
     epsilon
        [False, False, False],  #
     running_average_factor
    ]
    # Optional running_mean and
     running_var are only
    # connected to their new values.
    for i in range(5, len(node.inputs)):
        patterns[0].append(True)
        for pattern in patterns[1:]:
            pattern.append(False)
        patterns.append([False] * (3 + i
      - 5) + [True])
    return patterns
```

Table 1. Type 1 code clone example from the nnet module of the Aesara project

Snapshot code clone detection. We apply the following steps to every commit of the studied DL framework to detect clones. Firstly, we use the `git checkout` command to extract the framework snapshot corresponding to a specific commit from GitHub. Secondly, since in our study, we focus on code clones within production code, we follow the existing work [4] to exclude test-related code by identifying files

[13]https://www.txl.ca/txl-nicaddownload.html

```
def row(name = None, dtype = None):
    """Return a symbolic row variable (
     ndim=2, broadcastable=[True,False]).
    :param dtype: numeric type (None
     means to use theano.config.floatX)
    :param name: a name to attach to
     this variable
    """
    if dtype is None:
        dtype = config.floatX
    type = CudaNdarrayType(dtype=dtype,
     broadcastable=(True, False))
    return type(name)
```

```
def matrix(name = None, dtype = None):
    """Return a symbolic matrix variable
     .
    :param dtype: numeric type (None
     means to use theano.config.floatX)
    :param name: a name to attach to
     this variable
    """
    if dtype is None:
        dtype = config.floatX
    type = CudaNdarrayType(dtype=dtype,
     broadcastable=(False, False))
    return type(name)
```

Table 2. Type 2 code clone example from the cuda module of the Aesara project

```
def _find_bentoml_module_location():
    try:
        module_location, = importlib.
     util.find_spec(
            'bentoml'
        ).submodule_search_locations
    except AttributeError:
        # python 2.7 doesn't have
     importlib.util, will fall back to
     imp instead
        import imp

        _, module_location, _ = imp.
     find_module('bentoml')
    return module_location




    #
```

```
def _is_bentoml_in_develop_mode():
    try:
        module_location, = importlib.
     util.find_spec(
            'bentoml'
        ).submodule_search_locations
    except AttributeError:
        # python 2.7 doesn't have
     importlib.util, will fall back to
     imp instead
        import imp

        _, module_location, _ = imp.
     find_module('bentoml')

    setup_py_path = os.path.abspath(os.
     path.join(module_location, '..', '
     setup.py'))
    return not _is_pypi_release() and os
     .path.isfile(setup_py_path)
```

Table 3. Type 3 code clone example from the bundler module of the BemtoML project

and folders with the "test" keyword. Test code is typically meant for validation and verification of the system's functionality, rather than contributing directly to the production environment. Lastly, we use NiCad to detect the code clones. The output of the clone detection is pairs of functions, where two functions that are cloned, i.e., highly similar to each other, constitute a clone pair.

Clone genealogy generation. Clone pairs can undergo code changes throughout the development and maintenance stages of a framework. As a result of a change, a clone pair can maintain a *consistent* state, i.e., cloned status, or diverge to an *inconsistent* state. A genealogy represents the historical evolution of a pair of code clones. To generate the clone genealogy of each clone pair, we track its modification in every commit along the framework commit list. Similar to the existing work [4, 14], we conduct the below steps to generate the genealogy of every clone pair:

1. For a given commit C_i, we identify the files changed. If a changed file changes a clone pair, we proceed by comparing C_i to the previous commit C_{i-1} to verify if the change was made to the clone pair. We use the `diff` command and a python third party library *whatthepatch*[14] to map the lines corresponding to the beginning and end of the function in both commits C_i and C_{i-1}. To handle the case where a file name was modified, we map the renamed file to the original file by performing the following steps as introduced in the existing work [4, 14]. First, for each commit, we extract the pairs of newly added and deleted files between the current and preceding commit. Second, we compare the code similarity and consider that a file was renamed if the content of the new file is similar to the content of the old file. We use the below git command to construct the pairs of renamed files between two commits: `git diff [old-commit] [new-commit] -name-status -M`
2. We check if a code change, i.e., code lines addition or deletion, is made in C_i within the boundaries of the clone pair. If so, we verify if this clone pair exists in the clone snapshot corresponding to commit C_{i+1}. If the clone pair exists in C_{i+1}, we attribute the state *consistent* to it; otherwise, we consider the pair state as *inconsistent.*
3. We reiterate this process, i.e., steps 1 and 2, for every commit in the framework repository. This process continues until one of the following conditions is met: 1) once all commits in the framework repository have been processed, meaning that no new clones can be identified or existing ones updated; and 2) if one of the clones in an identified pair is deleted implying that the cloned relationship has been terminated.

Table 4. The Descriptive Statistics of the Dataset.

Framework	First commit	# of commits	# releases
Aesara	2008-01-06	29,600	14
Keras	2015-03-27	7,787	9
MXNet	2015-04-30	11,865	20
Ray	2016-02-07	16,403	19
PyTorch	2016-05-02	48,561	18
Paddle	2016-08-29	36,531	22
TensorFlow	2018-03-04	131,854	46
Jax	2018-11-17	13,868	7
BentoML	2019-04-01	2,100	11

2.4 Cross-framework file-level code clone extraction

To identify code clones across several DL frameworks, we employ SourcererCC [70], a token-based clone detection tool capable of identifying both exact and near-miss

[14]https://pypi.org/project/whatthepatch/

clones within extensive inter-framework repositories. We select SourcererCC as it outperforms other clone detectors, e.g., Nicad, for large-scale cross-projects clone detection [16, 65] and it was used by recent cross-project clone related work [16, 64, 84] for its high performance. We use the publicly available version[15] on GitHub. SourcererCC performs clone detection in two steps. Firstly, it employs a tokenizer where the programming language parameters, such as file extension, are set. In our case, we set the file extension to ".py". Secondly, the actual clone detection is executed. For this step, the similarity thresholds parameter is specified. Existing work uses different values for the similarity threshold. For example, Rahman et al. [64] use 70% whereas Chochlov et al. [10] use 50%. In our study, we vary the value between 60% and 80%. These values are chosen to achieve a balance between detecting file-level clones and avoiding overly restrictive or permissive thresholds. We vary the value between 60% and 80% based on an experimental setup, which shows that the 60% threshold is effective in detecting cloned files with substantial code segments and high structural and functional similarity across frameworks.

2.5 Metrics collection

After we generate the code clone snapshots and genealogies, we collect cloned metrics for every framework. The collected clone metrics capture information about the snapshot and the genealogy of a clone pair to investigate the relationship between the collected metrics and the cloned code size. In total, we collect 29 metrics, representing (1) product metrics, (2) development community metrics, (3) genealogy metrics and (4) code metrics. These metrics were used in existing work [4, 14, 74]. We describe below each of the categories and present the collected metrics in Table 5.

Product metrics. Product metrics are the attributes related to the snapshot, i.e., commit. The product metrics are collected from the snapshot of the framework that contains the clone pair. Example metrics include the number of siblings a clone has in the version where it is introduced, the size of the clone and the number of clone groups.

Development community metrics. Development community metrics include metrics representing the involvement of contributors in the development of clone pairs. For example, the number of committers, i.e., the person who applied the commit to the repository, and authors, i.e., the person who originally wrote the changes introduced in the commit, associated with the genealogy of clone pairs that provide insights into the collaborative efforts, i.e., number of individuals that have applied and integrated changes related to the clone pairs, and individual contributions to the codebase over time.

[15]https://github.com/Mondego/SourcererCC

Genealogy metrics. Genealogy metrics capture the state changes in the history of clone pairs and the history of changes in a clone pair. An example of a genealogy metric is the ratio of inconsistent changes in the whole framework code clone genealogy.

Code metrics. Code metrics include metrics related to the code characteristics of the method that contains a clone, such as cyclomatic complexity and the number of declaration and execution statements. To analyze code metrics, we use the Understand tool[16], a reverse engineering tool capable of analyzing product metrics from the source code of software systems. First, we run the Understand tool on all the snapshots of DL frameworks and extract the metrics on the function level. We obtain metrics such as the number of declarative statements in a function and the complexity of a function. Second, we map the functions obtained by the Understand tool to the clone code to compute the metrics for the cloned metrics.

2.6 Lifelong code cloning trends identification from DL frameworks

Our goal is to discern the lifelong evolution, i.e., long-term, trends of code clones within DL frameworks to gain insights into the historical and chronological changes in cloned code over multiple releases. This analysis can provide valuable information about the stability, growth, or reduction of code clones over time, aiding in the assessment of software evolution and maintenance practices within DL frameworks. For every DL framework, we build a time series, i.e., a temporal representation, capturing the historical and chronological changes to the cloned code over the releases of a framework. Then, we group similar timer series patterns together and identify the code clone trends over the releases. We elaborate on our approach in the following steps.

Step 1: Building time series data. To build the time series that represents the historical evolution of code clones, we start by calculating the clone coverage. Clone coverage represents the proportion of cloned code in relation to the overall codebase [23]. The clone coverage is calculated as follows:

$$\text{clone\_coverage}_i = \frac{\text{clone\_size}_i}{\text{SLOC}_i} \tag{1}$$

where $clone\_coverage_i$ represents the clone coverage corresponding for *commit i*. The $clone\_size_i$ represents the total number of lines of code belonging to clones at *commit i*. The $SLOC_i$ represents the lines of source code of the overall repository at *commit i*.

Our goal is to construct a release-level clone coverage time series for every framework. We choose the release-level because it represents a stable granularity as opposed to the commit-level granularity that would have been a fine granularity, given the small and/or temporary modifications that a commit can introduce [12]. We calculate the clone coverage for every revision, i.e., commit, using the code clone data extracted

[16]https://scitools.com/

Table 5. Collected code clone metrics.

Metrics	Description
	Product metrics
NumCP	The total number of clone pairs at any specific commit.
NumCPGroups	The unique number of groups at any specific commit.
MedianCLOC	The median number of cloned lines of code [4].
MaxCLOC	The maximum number of cloned lines of code [4].
MedianCSib	The median number of siblings of the clone pairs at any specific commit [14].
MaxCSib	The maximum number of siblings of the clone pairs at any specific commit [14].
CPMedianAgeDays	The median clone pair age in terms of the number of days [14].
CPMedianAgeCommits	The median clone pair age in terms of the number of commits [14].
NumUniqueFiles	The unique number of files that contain clone pairs.
NumAbstClasses	The number of abstract classes containing code clones.
MedCodeSim	The median of code clone similarity.
RatioSync	The ratio of clone pairs that have a final "consistent" change in the genealogy [4].
RatioLatePropCons	The ratio of clone pairs having late propagation, which end in a consistent change [4].
RatioLatePropIncons	The ratio of clone pairs having late propagation, which end in an inconsistent change [4].
	Development community metrics
NumCommittorsGen	The number of committers involved in the genealogy [14].
NumAuthorsGen	The number of authors involved in the genealogy
MedianNumComChanges	The median number of commits in the genealogy by a specific committer [14].
MedianNumAutChanges	The median number of commits in the genealogy by a specific author.
	Genealogy metrics
NumCPG	The number of clone pairs in the genealogy.
NumChanges	The total number of changes in the genealogy.
NumBursts	The number of change bursts on a clone. A change burst is a consecutive change with a maximum distance of one day between the changes [4].
NumInconsChanges	The number of inconsistent changes within the genealogy [4].
NumDiverChanges	The number of divergent changes (pattern "CI") in the genealogy [4].
NumResyncChanges	The number of resynched changes (pattern "IC") in the genealogy [4].
	Code metrics
MedianCompl	The median cyclomatic complexity of the cloned code [74].
MedianNumDeclStmt	The median number of declaration statements in a cloned code.
MedianNumExecStmt	The median number of execution statements in a cloned code.
MedianRatioCommentCode	The median ratio comment to code in cloned code [74].
MedianSumComplAbstMod	The median sum of cyclomatic complexity of all the classes containing cloned functions.

through the NiCad tool detailed in section 2.3 and the SLOC of the framework snapshot at the commit. Therefore, for every release R_j, we compute the median code clone coverage for all the commits within the time interval between two consecutive releases. We refer to the list of releases extracted from GitHub as described in section

2.2 and the associated release dates to isolate the commits falling within the release date intervals. Then, we calculate the median release clone coverage as follows:

$$\text{clone\_coverage}_{R_j} = \text{median}\left(\text{clone\_coverage}_i \text{ for } i \in R_j\right) \tag{2}$$

By the end of this step, we obtain the median value of code clone coverage for every release of every framework. For each DL framework, we construct a time series representing the evolutionary trajectory of clone coverage across multiple releases. By the end of this step, we obtain nine unique time series, one for each framework.

Step 2: Clustering time series. We utilize time series clustering to group time series presenting similar patterns in code cloning. This approach involves the selection of (i) a method for measuring distances, (ii) a clustering algorithm, and (iii) an optimal number of clusters, as we elaborate on below.

- *Dynamic Time Warping (DTW).* DTW is used as a distance measurement method [8] to align and measure the similarity between two time series that may have different temporal characteristics. In our context, it is used to compare and align the time series representing code clone coverage across multiple releases of DL frameworks. Different DL frameworks may have a distinct number of software releases, resulting in time series of varying lengths. To measure the similarity, i.e., distance, between time series data of different lengths, we adopt the DTW.

- *TimeSeriesKMeans clustering algorithm* is used for time series clustering [81] to identify patterns in the evolution of the nine DL frameworks. K-means clustering is a widely used unsupervised machine learning technique that is particularly effective for partitioning data into distinct groups or clusters based on the similarity of the data points. K-Means treats each time series as a point in a multidimensional space, aiming to group similar time series together in clusters. In our analysis, we employ tslearn [17] implementation of K-means for time series, TimeSeriesKMeans [18] to identify and categorize the trends present in the time series data representing code clone coverage within DL frameworks.

- *Optimal number of clusters.* Following prior work [60], we employ the silhouette score[66] to identify the most suitable number of clusters. The silhouette score assesses clustering quality by measuring how close data points are to their cluster compared to other clusters. The silhouette score ranges from -1 to 1, where a score close to 1 suggests well-defined clusters, and a score above 0.5 indicates a good clustering configuration. It leverages the output of the chosen clustering algorithm, e.g., K-Means, to determine the optimal number of clusters, aiming for the highest average silhouette score, which indicates well-separated

[17]https://tslearn.readthedocs.io/en/stable/gen_modules/tslearn.clustering.html#module-tslearn.clustering
[18]https://tslearn.readthedocs.io/en/stable/gen_modules/clustering/tslearn.clustering.TimeSeriesKMeans.html

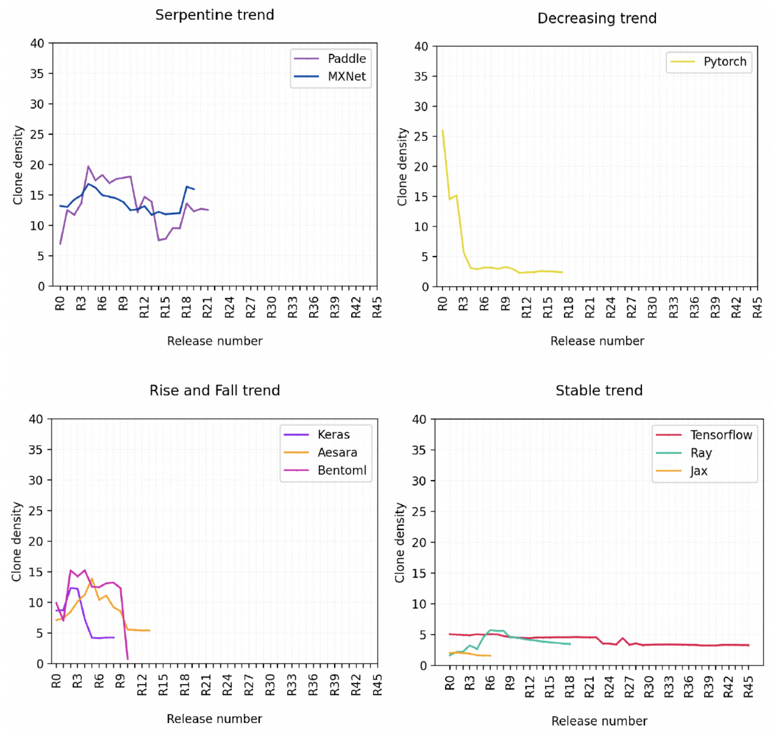

Fig. 2. The four code clones trends exhibited by DL frameworks.

and coherent clusters. We employ the silhouette score function[19] from tslearn. We use a range of numbers of clusters, k, between 2 and 8. As a result, we obtain 4 optimal numbers of clusters with a silhouette score of 0.63, suggesting a good quality of clusters. Additionally, we complement the quantitative result with human judgement. The first two authors of the paper manually examine the clustered code clone trends to ensure that the automated clustering algorithm produces results that are practically coherent.

Our approach identifies four trends, i.e., *"Serpentine"*, *"Rise and Fall"*, *"Decreasing"*, and *"Stable"*, in the evolution of clones observed in DL frameworks as illustrated in Figure 2. We describe the features of every pattern below:

– The *"Serpentine"* trend reflects a cyclic, oscillatory pattern in code clones, where coverage alternates between highs and lows, presenting fluctuations in code clone coverage over successive releases.

[19]https://tslearn.readthedocs.io/en/latest/gen_modules/clustering/tslearn.clustering.silhouette_score.html

- The *"Rise and Fall"* trend marks intermittent rises in clone coverage followed by an overall prolonged decrease, resulting in a downward trajectory over time.
- The *"Decreasing"* trend represents a consistent and continuous reduction in clone coverage across successive releases.
- The *"Stable"* trend exhibits a relatively constant and consistent clone coverage with minimal fluctuations over releases.

For example, while *Paddle* and *MXNet* display a *"Serpentine"* trajectory, *BentoML*, *Keras*, and *Aesara* follow a *"Rise and Fall"* trajectory that shows a prolonged decrease despite intermittent rises. *PyTorch* exhibits a consistent decreasing clone coverage, i.e., starting at 27% at the first release and dropping to less than 3% after 20 releases. *TensorFlow*, *Jax*, and *Ray*'s clone coverage remain relatively stable, with minimal fluctuations observed across all releases remaining within a narrow range (less than 5%).

2.7 Within-release development patterns identification

Our goal is to identify the within-release development patterns, i.e., the patterns of cloned code size evolution from one commit to another within a framework release. This analysis investigates how code clones evolve within each individual release and can help uncover the impact of the within-release patterns on the long-term code cloning trends. Our approach involves two key steps:

Step 1: Building within-release cloned code size time series. The evolution of code cloned code size between the commits of a release can be interpreted as time series data. To build the time series for every release of every framework, we select the commits that belong to every release interval by referring to the release date. Then, we calculate the cloned code size using the code clone data extracted through the NiCad tool detailed in section 2.2 for every commit. In total, we obtain 153 time series corresponding to the nine DL frameworks' releases.

Step 2: Clustering within-release code clone time series. Similarly to the clustering step in section 2.6, we leverage DTW and TimeSeriesKMeans to cluster the obtained time series. We determine the number of clusters using the silhouette score, and we obtain 3 as the optimal number of clusters, giving a silhouette score of 0.66, indicating a high quality of clusters. Figure 3 shows a subset of the time series clustered into the three *"Ascending"*, *"Descending"*, and *"Steady"* patterns.

The analysis of release-level time series reveals three distinct patterns, i.e., *"Ascending"*, *"Descending"*, and *"Steady"*. As depicted in Figure 3, the *"Ascending"* pattern signifies a consistent increase within a release, the *"Descending"* pattern reflects a continuous decrease within a release, and the *"Steady"* pattern suggests a relatively constant behaviour within a release. Table 6 shows the distribution of time series

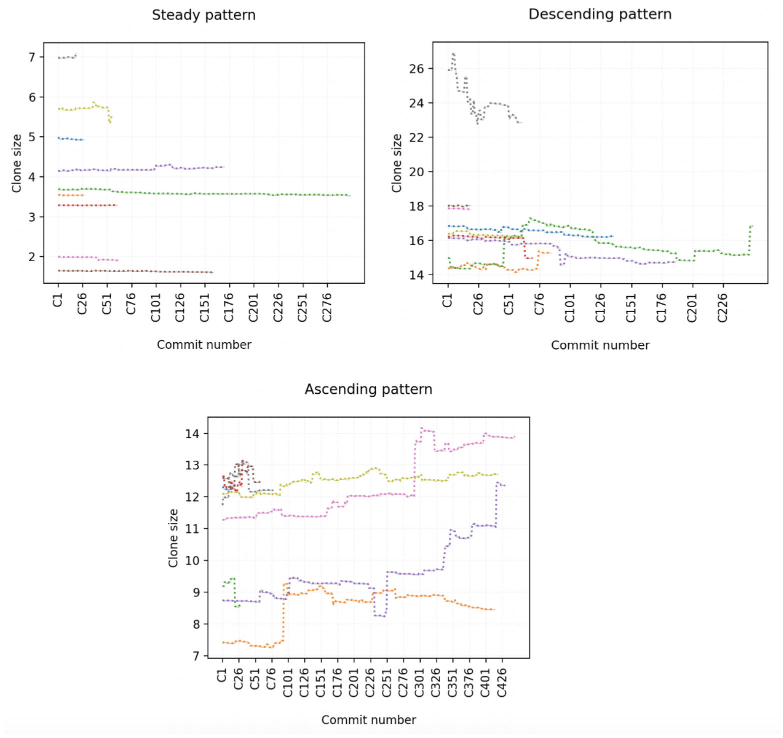

Fig. 3. Subset of the within-release time series *"Steady"*, *"Descending"* and *"Ascending patterns"*. Every time series represents the evolution of clone size within a particular release of a DL framework.

Table 6. The distribution of release-level time series across the three within-release code cloning patterns.

Pattern	**TensorFlow**	**Paddle**	**PyTorch**	**Aesara**	**Ray**	**MXNet**	**Keras**	**Jax**	**BentoML**	**Total**
Ascending	0	11	0	8	0	12	4	0	4	39
Descending	0	8	4	0	0	9	0	0	4	25
Steady	38	2	14	4	19	0	4	5	3	89

among these patterns.

3 Results and Analysis

In this section, we identify the different code clone evolution trends in DL frameworks over multiple releases and investigate the within-release development practices shaping the release-level clone trends. In addition, we study the cross-framework clones within DL frameworks. Precisely, we discuss motivation, approach and findings for the following research questions.

3.1 RQ1: What are the characteristics of the long-term trends observed in the evolution of code clones within DL frameworks over releases?

3.1.1 Motivation

The field of DL is marked by rapid growth and continual advancements. As DL frameworks rapidly evolve, they are prone to technical debts [29, 77] and defects, making the system harder to maintain. For example, code segments with defects could be cloned by developers without the knowledge of the defects. This could lead to risks of using the cloned code. Recent work [29, 46] demonstrates that code clones are prevalent in DL applications. In this RQ, we aim to investigate the characteristics of code clone evolution, such as bug-proneness and community activity, and explore how these characteristics change over time. By examining these aspects, we seek to provide a deeper understanding of how the evolution of code clones affects code quality and maintainability in the fast-evolving landscape of DL frameworks.

3.1.2 Approach

Our goal is to explore the characteristics associated with the identified code clone trends. We study the characteristics of the trends along three dimensions: cloned code size trend, bug-proneness and community size.

Investigating the decrease of clone coverage. To better understand the rationale behind the decrease in clone coverage in the long-term descending trends, i.e., trends that exhibit a lower long-term clone coverage, we conduct a manual analysis. First, we investigate the evolution of the cloned code size, i.e., the number of lines of cloned code, associated with the median data point of each release to check if the decrease in clone coverage is associated with a decrease in the cloned code size. Then, for each release, we manually read the release notes and the commit messages of the commits associated with the release interval, thereby gaining insights into the rationale of the evolution of clone coverage over time.

Investigating bug-proneness within trends. We follow a two-step approach to study the bug-proneness of code clones. First, we apply the keyword-based heuristic introduced by Mockus et al. [47] and adopted in related work [5, 30, 46, 49] to identify bug-fixing commits. We consider a commit as a bug-fixing commit if the commit message contains any of the keywords *bug, fix, wrong, error, fail, problem, and patch*. Second, we match the bug-fixing commits to the cloned code. Specifically, we employ the `git diff` command to retrieve the altered lines, i.e., added or deleted, within each bug-fixing commit and compare them to the clones identified by NiCad. If the modified lines for fixing a bug occurred within a clone code, we classify that clone as potentially susceptible to bugs. This approach allows us to pinpoint the bug-fixing commits occurring in clones. After identifying the bug-fixing commits belonging to the cloned code, we plot the evolution of the bug-proneness of cloned code over releases. To quantify bug-proneness, we compute bug-fix density$_i$, i.e., the ratio of bug-fixing commits in the cloned code introduced within a release interval by the total number of all commits to code clones introduced within a given release *i*. This ratio is calculated as follows:

$$\text{bug-fix density}_i = \frac{\delta\,(\text{clone\_bugs\_fix\_commits}_i)}{\delta\,(\text{all\_clone\_commits}_i)} \tag{3}$$

where $\delta(\text{clone\_bugs \_fix \_commits}_i)$ represents the commits introduced within release *i* interval and classified as bug-fixing and $\delta(\text{all\_clone \_commits}_i)$ represents the total number of commits that occurred in cloned code within the release $_i$. A bug-fix density ratio of 0 signifies no bug-fixing commits in the cloned code during a release, while a ratio of 1 indicates that all commits within that release for the cloned code are bug-fixing commits. By tracking the evolution of this ratio across releases, we obtain insights into the dynamic relationship between code clone trends and bug-proneness, enhancing our understanding of code quality and maintenance over time.

Investigating bug-proneness in "thin clones" vs. "thick clones." Thin clones refer to code clones that involve a smaller number (i.e., 7 lines of code) of duplicated lines of code, while thick clones represent clones with a larger number (i.e., 10 lines of code) of duplicated lines. We identify "thin clones" and "thick clones" based on the distribution of clones based on the percentiles. We identify *"thin clones"* and *"thick clones"* based on the distribution of clones based on the percentiles. Firstly, we gather all the clone snapshots from all the frameworks. For each snapshot, the median cloned code size is computed, taking into account the diverse sizes of clone pairs within each snapshot of the codebase. Subsequently, the 25^{th} and 75^{th} percentiles are calculated, establishing thresholds for *"thin clone"* and *"thick clones"*, respectively. More specifically, code clones falling below the 25^{th} percentile are categorized as *"thin clone"*, while those exceeding the 75^{th} percentile are labelled as *"thick clone"*. Following the identification of bug-fixing commits, we map each change to the corresponding cloned code (as described in previous steps). Through this mapping, we categorize each commit-fix as either fixing a *"thin clone"* or a *"thick clone"*. To assess the statistical significance of bug-proneness differences between "thin clones" and "thick clones," we employ a Mann-Whitney U test on the percentage distribution of the number of changes to cloned code in *"thin"* and *"thick"* clones. This non-parametric test is chosen due to the potential non-normality of the data and its suitability for comparing two independent samples. If we obtain a p-value <= 0.05, we reject the null hypothesis and conclude that the distributions of bug-fixing percentages in the two code clone categories are different. Studying bug-proneness in "thin clones" versus "thick clones" provides insights into whether bugs tend to concentrate more on one type of clone over the other, which can help understand how code clones impact software quality and maintenance.

Investigating the bug impact in clones. Different trends of code clones may exhibit diverse characteristics, such as cloned code size and distribution of thick and thin clones, which could influence the bugs' impact within the codebase. To investigate the extent of bug impact within code clones, we examine the number of file changes per bug-fixing commit. This metric allows us to assess the extent to which bugs spread in the codebase. The goal is to discern potential differences in bug impact across various trends. Specifically, we evaluate whether the bug impact varies significantly among different trends. We formulate the following null hypothesis: *H01: Different trends of code clones exhibit different bug impact trends.* To assess significance, we employ Welch's ANOVA test from scipy library [20], a statistical analysis method suited for comparisons of groups with unequal sizes. Additionally, we also compare bug impact, i.e., by examining the number of file changes per bug-fixing commit, in cloned and non-cloned code. This comprehensive approach allows us to gain insights into the distinctive patterns of bug impact within code clones and explore potential variations in comparison to non-cloned code.

Investigating the community size evolution in clones. The community size is used to measure the proportion of contributors, i.e., authors of the code, engaged in clone-related activities within DL frameworks relative to the total contributors involved in the entire codebase. The community size represents an indicator of the collaborative involvement of contributors to clones. To derive the community size evolution, we employ the following steps. First, for each release, we calculate the number of contributors engaged in clone-related activities up to the release date. Subsequently, we determine the community size by dividing the number of contributors involved in clones by the total count of contributors to the entire codebase up to the respective release date. This metric provides a measure of how the number of contributors in clone-related efforts evolves over the course of the framework.

Investigating clone types distribution and evolution. To gain a deeper understanding of code clone dynamics in DL frameworks, we investigate the evolution and distribution of clone types (i.e., Type 1, Type 2, and Type 3 clones). By examining the clone types we can identify associations in code reuse and duplication that may be indicative of specific coding practices and code cloning trends. For the distribution analysis, we calculate the percentage of clone pairs belonging to each type for every release of each DL framework. We visualize, using box plots, the variability of clone types distributions across frameworks. To study the evolution of clone types over time, we track the changes in the proportion of each type across different releases for each framework.

[20]https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.f_oneway.html

3.1.3 Results

The decline in overall clone coverage observed in the *"Decreasing"* and *"Rise and Fall"* trends can be attributed to 1) a decrease in the cloned code size or a slow increase in the number of cloned code size compared to a rapid expansion of the codebase. We conduct a comparison of cloned code size at the initial and final releases of each framework. As shown in Table 7, we observe that BentoML presents a reduction in cloned code size over time, resulting from a decrease in clone groups and clone siblings within a clone group. For example, despite a more than twofold increase in framework size between the first and last releases, BentoML's cloned code size decreased from 982 to 174 lines of code. This decrease in cloned code size corresponds to a decrease in the clone groups from 23 to 5. Hence, the decrease in clone coverage of BentoML from 10% in the first release to less than 1% in the last release. Aesara, Keras, and PyTorch also exhibit a decrease in clone coverage between the first and last release. Specifically, the clone coverage for Aesara, Keras, and PyTorch drop from 7% to 5%, 9% to 4% and 26% to 2%, respectively, between the first and last release. However, the decrease for the three frameworks is attributed to the rapid expansion of the codebase, outpacing the growth rate of the cloned code size. For example, while the codebase for Pytorch increased by a factor of 30 between the first and last release, the size of the clone code only increased by less than threefold.

The decline in cloned code size can be attributed to code refactoring, third-party library reuse, and code clone removal associated with feature elimination. To understand the factors contributing to the decrease in clone code density, we conduct an inter-release manual analysis by reading the release notes and commit messages for releases demonstrating a reduction in cloned code size compared to their preceding release. We find that cloned code size reduction is due to 1) code refactoring, 2) substitution of code fragments due to the use of third-party libraries, and 3) the removal of clone code as a consequence of eliminating certain features. For instance, duplicate code is moved to a shared utility code. For example, in release *2.2.0*[21], the Keras framework underwent a large code refactoring[22] that affected the clones. In another release *2.4.0*[23] of the Keras framework, the cloned code size is reduced due to a redirect of all APIs in the Keras package to tf.keras third party-library. In the release *2.2.0*[24] of the Aesara framework, clones are eliminated as a result of the elimination of the function *local_neg_neg* from the tensor module.

Bug fixing is a persistent activity consistently occurring throughout the lifespan of frameworks, among all the code cloning trends. The *"Serpentine"* trend is more susceptible to bugs. Figure 4 represents the evolution of bug-proneness

[21]https://github.com/keras-team/keras/releases/tag/2.2.0
[22]https://github.com/keras-team/keras/pull/10865/commits
[23]https://github.com/keras-team/keras/releases/tag/2.4.0
[24]https://github.com/aesara-devs/aesara/releases/tag/rel-2.2.0

Table 7. Cloned code size analysis of the DL frameworks exhibiting "Decreasing" and "Rise and Fall" trends from first to last Release.

	Metric	**First release**	**Last release**	**Trend**
BentoML	SLOC	9,917	22,494	↗
	Code size (SLOC)	982	174	↘
	Clone coverage	0.1	0.008	↘
	# of siblings	58	18	↘
	# of groups	23	11	↘
Aesara	SLOC	39,021	84,773	↗
	Code size (SLOC)	2,765	4,612	↗
	Clone coverage	0.07	0.05	↘
	Total number of clone siblings	307	396	↗
	Total number of clone groups	117	139	↗
Keras	SLOC	19,382	139,816	↗
	Code size (SLOC)	1,680	5,947	↗
	Clone coverage	0.09	0.04	↘
	# of siblings	117	335	↗
	# of groups	43	98	↗
PyTorch	SLOC	10,567	313,603	↗
	Code size (SLOC)	2,741	7,527	↗
	Clone coverage	0.26	0.02	↘
	# of siblings	275	583	↗
	# of groups	62	202	↗

in clone code in DL frameworks over releases. When comparing bug-proneness percentages across frameworks, we notice distinct variations. For instance, frameworks belonging to the *"Serpentine"* trend, i.e., *Paddle* and *MXNet*, demonstrate higher bug-proneness percentages across releases in comparison to the frameworks belonging to the *"Decreasing"*, *"Rise and Fall"* and *"Stable* trends. For example, *Paddle* and *MXNet* frameworks have 50% and 75% of the releases having more than 50% bug-fixing commits in clones respectively, whereas in *TensorFlow* all 100% of the framework releases have less than 50% of the bug-fixing commits in clones as shown in 4. ANOVA-Welch test indicates a significant difference between the bug-proneness evolution of *Paddle* and *MXNet* frameworks and the frameworks belonging to the other three trends. These findings suggest that the DL frameworks belonging to the *"Serpentine"* trend characterized by a fluctuating code clone trend could be more susceptible to bugs.

Moreover, the *Keras* and *Aesara* frameworks belonging to the *"Rise and Fall"* clone trend exhibit a decreasing bug-proneness trend where the overall bug-proneness percentage decreases from approximately 40% in initial releases to almost zero in subsequent releases. For the Aesara framework, we notice a spike to 100% in the bug-proneness at release 12, meaning that all of the commits introduced were fixing bugs. Further investigation demonstrates that the 100% rate is due to a short release cycle of 11 days, during which only one commit to a clone occurs, and this solitary commit

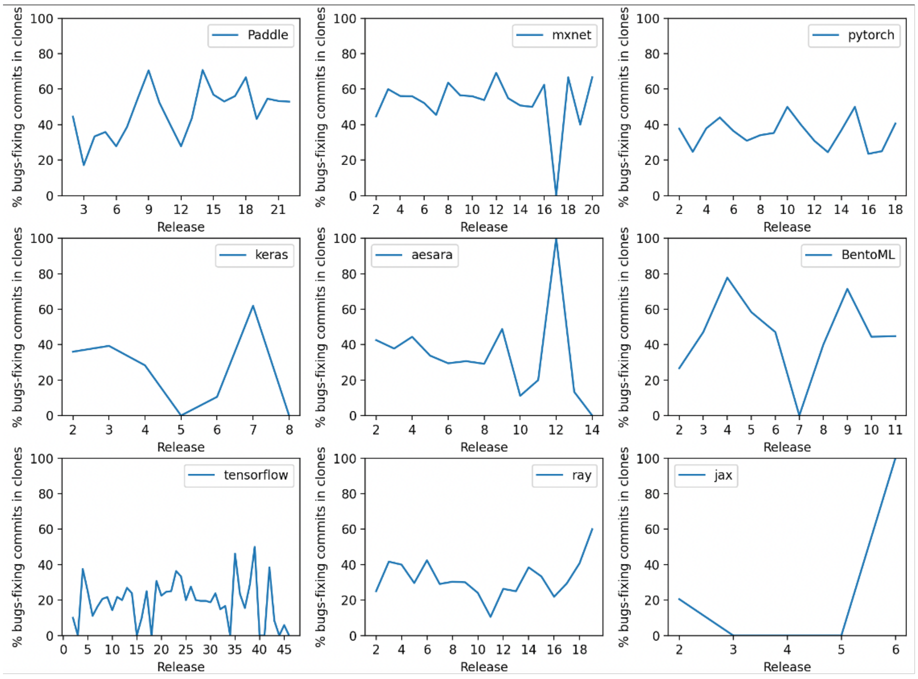

Fig. 4. The bug-proneness evolution in cloned code over releases.

happens to be a bug fix. Similarly, the abrupt decrease in bug-proneness to zero in the *MXNet* framework at release 17 is linked to the absence of commits to code clones during a relatively short release cycle of 27 days. The observed descending trends in code clones, coupled with a corresponding reduction in bug-proneness, suggest an enhancement in overall code quality and reliability. These findings underscore the importance of investigating the coding practices followed within the release and their impacts on the code clone in DL frameworks, which we address in the following research question.

Over 50% of the changes implemented in bug-fixing commits predominantly occur within *"thick"* cloned code as opposed to *"thin"* cloned code across all the long-term code clone trends. Figure 5 shows the distribution of changes across the frameworks. The Mann-Whitney U test reveals a significant difference in bug-proneness between *"thin* clones and *"thick"* clones with p-value < 0.05. In addition, we investigate if the difference among the clone categories is of strong significance by calculating the Common Rank (Z) score. We obtain a Z value of 3.57 that suggests a substantial and statistically significant difference between the two bug-fixing commits in *"thick"* and *"thin"* code clones. A z-score [25] of +/- 1.96 or greater is considered statistically significant at the 5% level of significance (i.e., p < 0.05). Additionally, we

[25]https://www.z-table.com/

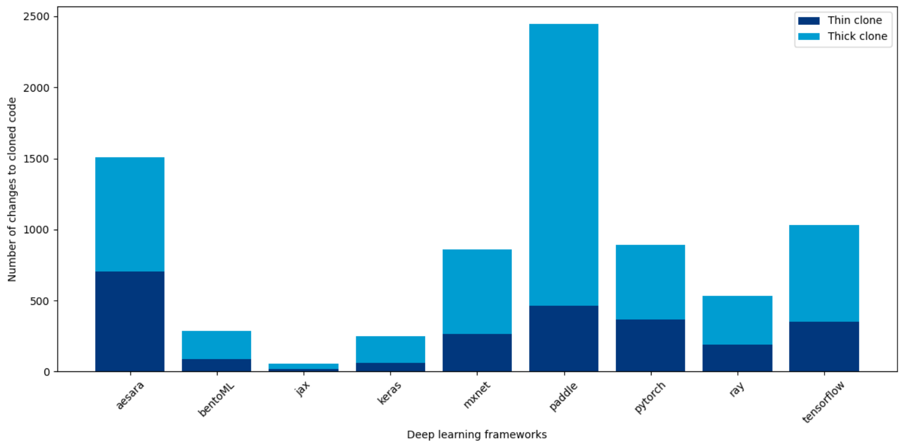

Fig. 5. Bug-fixing commits distribution by *"thin clones"* and *"thick clones"*.

employ the Cliff's Delta as another measure of effect size. A value of 1 indicates a large effect size. We obtain a Cliff's Delta value of 1, which further reinforces the notion that the bug-proneness difference between *"thin"* and *"thick"*. These results provide statistical evidence supporting the assertion that bug-proneness varies significantly between *"thin* clones and *"thick"* clones in our study. This result highlights a substantial distinction in bug-fixing tendencies between the two clone categories with *"thick"* clones being more susceptible to bug-fixing commits.

Our analysis reveals no significant difference in the number of files changed per bug-fixing commit across long-term code clone trends, as validated by Welch's ANOVA test, and we reject the null hypothesis. However, comparing the bug impact between cloned and non-cloned code across the frameworks indicates a significant difference in the number of files changed per bug-fixing commit in clones versus non-clones. Specifically, non-cloned files exhibit a higher number of changes, with an average of 2.4 files changed per commit, compared to 1.5 files changed per commit in cloned code. This could suggest that bug fixes in cloned sections are more localized and targeted.

The community involvement in clone activities remains consistent throughout the lifetime of the frameworks, with exceptions in *BentoML* and *Ray*, where original authors dominate clone maintenance and the community size witnesses a decrease over time. For instance, a relatively small portion of the community actively contributes to clones, consistently amounting to less than 50% for the majority of releases. Figure 6 depicts the community size evolution, i.e., the percentage of contributors to clones across various DL frameworks. As we can notice, the percentage of the community involved in clone-related presents a stable and sustained level of

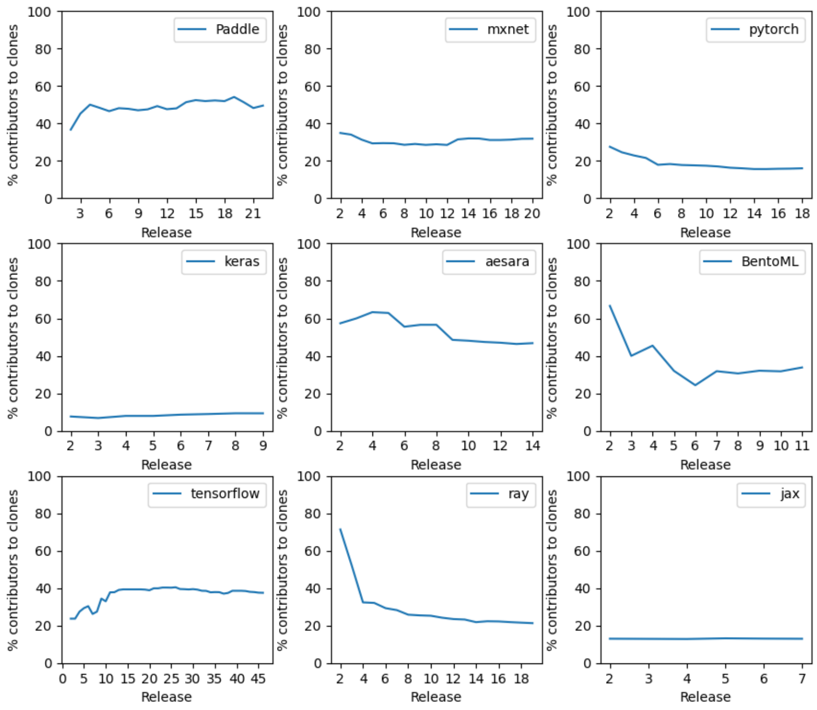

Fig. 6. The evolution of code clone community size over releases.

contribution to clones over time, among all the long-term cloning trends of the framework. However, there is a distinct reduction in community involvement in code clones within the frameworks *BentoML* and *Ray*. This decline in community participation can be attributed to the circumstance where the contributors making changes to clones are predominantly the original authors of those clones. To investigate this observation further, we manually check the clone pairs within *BentoML* and *Ray*. We notice that for those frameworks, the original creators of the clones are the ones maintaining them.

The evolution of clone types varies across different DL frameworks, suggesting that the factors influencing code reuse and duplication are shaped by the unique features and design choices of each framework. Figure 7 illustrates the distribution of clone pair types (i.e., Type 1, Type 2, and Type 3) across the various DL frameworks. Each box plot shows the variability in the code clone type percentages across the releases of a particular DL framework, indicating the distribution of clone types within that framework over time. First, we observe that Type 3 clones are dominant across most frameworks. The higher median of Type 3 clones represents a dominance of the clone type which indicates that developers often modify code rather than creating exact duplicates (Type 1) or structurally similar code (Type 2). This

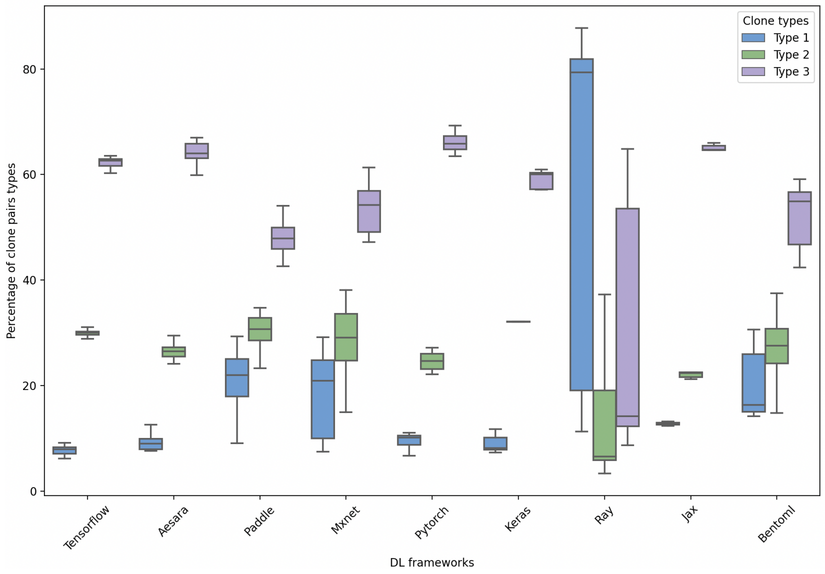

Fig. 7. The variability in the distribution of clone types (clone pairs) across all releases of the DL frameworks

reflects the need for customization and adaptation in DL frameworks and a potential increased maintenance effort since managing Type 3 clones can be more challenging than Type 1 or Type 2 clones because the variations within the clones mean that changes to one instance may not easily propagate to others. Second, we notice that clone types present a small variability in the distribution within frameworks with the exception of Ray project that presents a high variability in Type 1 and Type 3 clones. The broad range of Type 1 and Type 3 clones suggests a fluctuating level of code duplication (Type 1) and code modification (Type 3) across releases. Figure 8 represents the evolution of code clone types over multiple releases. The evolution of clone type distributions varies across different DL frameworks. For frameworks like TensorFlow, Aesara, PyTorch, Jax, and Keras, the distribution of clone types remains relatively stable, indicating consistent patterns of code reuse and duplication. However, other frameworks, such as Ray, exhibit increasing or decreasing trends in certain clone types. For instance, Ray shows an increase in Type 1 clones in later releases, suggesting a potential trend towards more straightforward and less complex code duplication. Our analysis reveals no direct association between code cloning trends and the evolution of clone types. This suggests that factors such as framework design, community practices, and project-specific requirements, play a role in shaping the dynamics of code reuse and duplication in DL frameworks.

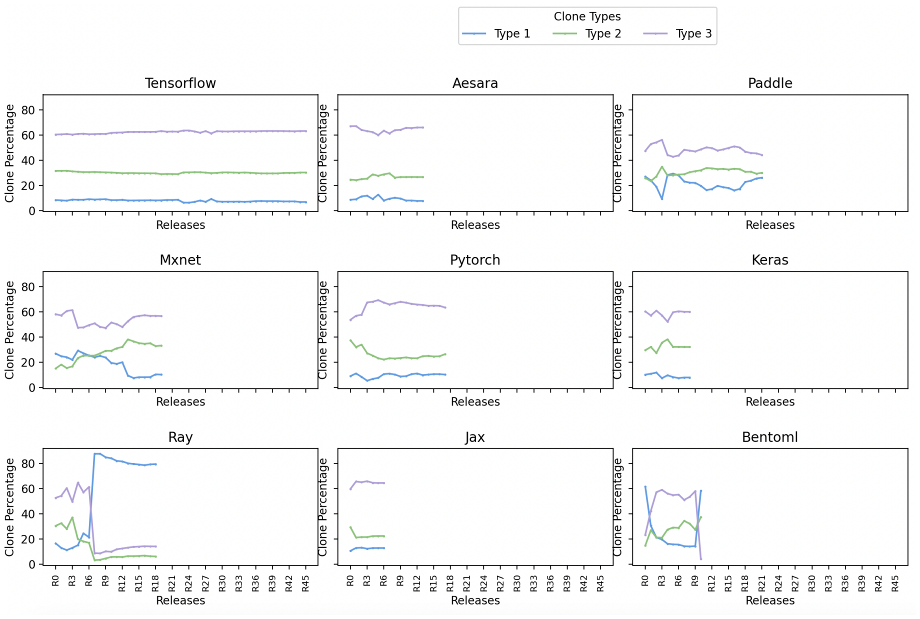

Fig. 8. The evolution of clone types in DL frameworks

Summary of RQ 1

Long-term code cloning trends exhibit some common and distinct characteristics. The decline in overall clone coverage observed in the *"Decreasing"* and *"Rise and Fall"* trends is attributed to 1) a decrease in the cloned code size or 2) a slow increase in the cloned code size compared to a rapid expansion of the codebase. All long-term trends are prone to bugs, and the bug-fixing activities are consistent across the lifespan of the frameworks of all the trends. However, the framework belonging to the *"Serpentine"* trend, i.e., *Paddle* and *MXNet*, could be more susceptible to bugs as they present a higher percentage of bug-proneness across the releases indicated by a higher commit-fixing. *"Thick"* clones present a higher bug-proneness as compared to *"thin"* clones across all the long-term trends. Regarding the community involvement in clone activities, it remains consistent through the lifetime of the frameworks, with exceptions in *BentoML* and *Ray*, where original authors dominate clone maintenance and the community size witnesses a decrease over time.

3.2 RQ2: What are the characteristics of within-release development patterns and do these patterns contribute to the overarching long-term trends in code cloning?

3.2.1 Motivation

In RQ1, we have investigated the characteristics of long-term clone trends over releases and observed that every trend has distinct characteristics, such as high susceptibility of the *"Serpentine"* trend to bug-proneness. In this RQ, we are interested in investigating the short-term clone patterns within each individual release. Our goal is to discern the impact of these patterns on long-term cloning trends and identify the characteristics of the within-release code cloning patterns. More specifically, we want to understand the factors influencing the evolution of cloned code size and provide insights into the dynamics of development practices within individual releases. In addition, the gained knowledge about within-release patterns, when aggregated over multiple releases, could contribute to understanding how development practices within releases influence clone evolution trends over time. By uncovering these relationships, we seek to formulate insights to enhance the efficiency and maintainability of DL frameworks within the code clone context.

3.2.2 Approach

After we obtain the three within-release code cloning patterns *"Ascending"*, *"Descending"*, and *"Steady"*, we aim to obtain a chronological evolution of how the within-release pattern contributed to the long-term trends. Therefore, we construct, for every DL framework, the evolution sequence of within-release patterns over the framework releases. Table 8 depicts the obtained chronological pattern sequences. P_a, P_d and P_s denotes an *"Ascending"*, *"Descending"*, and *"Steady"* pattern respectively.

Our approach to building the regression models comprises the following three steps:

- **Step1: Correlation and redundancy analysis of the independent variables.** we collect 37 metrics as detailed in section 2.5. The presence of correlated metrics might affect the performance of the model [33], therefore, we apply varclus[26] function in R to detect the existence of collinearity and exclude one metric of any pair of metrics that achieves a coefficient of 0.7 [45]. Then, we calculate the Variance Inflation Factors (VIFs) using the VIF[27] function in R for each independent variable to detect multicollinearity issues. We exclude all independent variables with VIF values above 5 because it indicates the existence of multicollinearity [11, 82].

- **Step 2: Constructing the models.** We construct three different regression models for each pattern shown in Figure 3: $Model_{Ascending}$, $Model_{Descending}$, and $Model_{Steady}$. For a $Model_i$, we assign to each snapshot revision, i.e., commit, the

[26]https://search.r-project.org/CRAN/refmans/Hmisc/html/varclus.html
[27]https://www.rdocumentation.org/packages/regclass/versions/1.6/topics/VIF

value 1 if the associated within-release pattern is i and 0 otherwise. For example, for $Model_{Ascending}$, we assign 1 to the commits that are clustered in the release belonging to the *"Ascending"* pattern. Since our dependent variable is binary, we follow the existing work [35, 58, 83] and employ logistic regression models. In particular, we fit a binomial logistic regression model using the glm[28] function provided by the state R package.

– **Step 3: Evaluating the models and assessing the significance of the independent variables.** After constructing the models, we use the Area Under the Receiver Operating Characteristic Curve (AUC) [40] to assess the predictive power of the created logistic models. A predictive model is deemed promising if the AUC-ROC is 0.7 or higher, with 1 signifying perfect predictive power[17, 19]. Next, we employ the ANOVA test [61] to assess the significance, measured in terms of (χ^2) for each independent variable in our model. The (χ^2) values for each variable are calculated as a percentage of the total (χ^2) values for all variables. We use upward and downward arrows to signify direct and inverse relationships between independent and dependent variables, respectively.

3.3 Results

Long-term descending trends, i.e., *"Decreasing"* and *"Rise and Fall"*, in addition to *"Stable"* trends consistently exhibit *"Steady"* within-release patterns. As we notice in table 8, the frameworks belonging to the long-term descending trends, i.e., *Keras*, *Aesara*, *BentoML*, *PyTorch*, *JAX*, *Ray* and *TensorFlow* are characterized by a consistent and consecutive repetitive stable pattern P_S. These stable patterns constitute over 30% of the total within-release patterns of each framework. This observation shows a connection between the two temporal scales and how the patterns of code clones within releases, i.e., short-term, have long-term implications. Hence, we proceed next with understanding the characteristics of the within-release code clone development practices.

The $Model_{Descending}$ and $Model_{Steady}$ demonstrate the most robust explanatory capability among the three constructed models. Table 10 provides an overview of our created models. The logistic regression models for $Model_{Descending}$ and $Model_{Steady}$ exhibit the highest AUC values at 0.92, whereas $Model_{Ascending}$ achieved an AUC of 0.89. Each model highlights a distinct attribute with the greatest explanatory strength, as shown in Table 9 (due to the space constraint, the table shows only the top five significant features for all three constructed models). Further insights into the analysis of these constructed models are detailed below.

"Ascending" code cloning pattern analysis

– **The decreased involvement of committers in the clone pairs clone genealogy is associated with a larger cloned code size**. As shown in Table 9,

[28]https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/glm

Table 8. Within-release patterns. P_a, P_d and P_s denote an *"Ascending"*, *"Descending"*, and *"Steady"* pattern respectively.

DL framework	**Within-release chronological patterns**
MXNet	$P_a, P_a, P_a, P_d, P_d, P_d, P_d, P_d, P_d, P_d, P_a, P_a, P_a, P_a, P_a, P_a, P_a, P_a, P_d, P_d$
Paddle	$P_s, P_a, P_a, P_a, P_d, P_d, P_d, P_d, P_d, P_d, P_d, P_a, P_d, P_a, P_s, P_a, P_a, P_a, P_a, P_a, P_a$
Keras	$P_a, P_a, P_a, P_a, P_s, P_s, P_s, P_s$
Aesara	$P_a, P_a, P_a, P_a, P_a, P_a, P_a, P_a, P_s, P_s, P_s, P_s$
BentoML	$P_a, P_a, P_d, P_d, P_d, P_d, P_a, P_a, P_s, P_s, P_s$
PyTorch	$P_d, P_d, P_d, P_d, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s$
JAX	P_s, P_s, P_s, P_s, P_s
Ray	$P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s$
TensorFlow	$P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s,$ $P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s, P_s$

Table 9. A summary of the analysis of the constructed three regression models.

Metrics	**Coef.**	χ^2**(%)**	**Pr(<**χ^2**)**	**Sign.**	**Relationship**
			$Model_{Ascending}$		
# of committers to a clone pair	-3.62	39.67	$< 2.2e^{-16}$	***	$\searrow$
complexity of abstract module	7.74	23.90	$< 2.2e^{-16}$	***	$\nearrow$
size of cloned code (med)	4.98	21.56	$< 2.2e^{-16}$	***	$\nearrow$
# clones siblings (max)	-5.30	16.99	$< 2.2e^{-16}$	***	$\searrow$
clone life (# commits)	9.50	4.85	$< 2.2e^{-16}$	***	$\nearrow$
			$Model_{Descending}$		
# changes to clone pairs	-9.08	24.86	$< 2.2e^{-16}$	***	$\searrow$
complexity of abstract module	-18.58	24.63	$< 2.2e^{-16}$	***	$\searrow$
size of cloned code (med)	8.00	23.83	$< 2.2e^{-16}$	***	$\nearrow$
similarity	-5.18	14.58	$< 2.2e^{-16}$	***	$\searrow$
cyclomatic complexity	3.78	11.94	$< 2.2e^{-16}$	***	$\nearrow$
			$Model_{Steady}$		
# of declaration statement (med)	-8.54	36.56	$< 2.2e^{-16}$	***	$\searrow$
# abstract classes (med)	4.36	25.61	$< 2.2e^{-16}$	***	$\nearrow$
# changes to clone pairs	3.34	18.08	$< 2.2e^{-16}$	***	$\nearrow$
# of clones siblings (med)	45.89	6.85	$< 2.2e^{-16}$	***	$\nearrow$
# of unique file	-2.70	5.78	$< 2.2e^{-16}$	***	$\searrow$

the feature number of committers to a clone pairs accounts for the highest (χ^2) in the $Model_{Ascending}$, suggesting that the fewer is the number of developers changing a clone pair, the higher is the likelihood of cloned code size increase. This could be explained by the fact that fewer committers modifying the clone pairs know the code well and are propagating the changes while maintaining the code clones. When new committers make changes to existing clones, the probability of applying the changes to all clones' copies decreases as they might not be aware of its presence, leading to inconsistent copies, hence a decreased cloned code size. For example, in the *PyTorch* framework, the cloned functions *acc_ops_max_pool2d* and *acc_ops_avg_pool2d* of the converter module were modified consistently and maintained the consistent state through the three commits *239b38268b*[29],*0d8a8a2e41*[30] and *e23827e6d6*[31] by the same committer. Whereas, in another instance of the same framework, the optimizer *AdagradOptimizer* and *AdamOptimizer* that were clones were modified independently. A first commit *05542f6222*[32] made by one committer made changes to *AdagradOptimizer* only leading to an inconsistent state, then another commit by another committer *812bc1dde6*[33] propagated the change to the second optimizer, which led a consistent state again. **In addition, we highlight that the positive correlation (in the same model) between clone pairs' longevity (i.e., clone life in terms of the number of commits and the increased cloned code size) suggests that the clone pairs are either changed consistently or not modified**.

– **Code clones size tends to increase when abstract classes exhibit higher complexity**. Cloned code size positively correlates to the complexity of the cloned code in abstract classes. It is the second most explanatory factor in $Model_{Ascending}$. The rationale is likely rooted in the observation that code cloning often occurs under conditions of higher code complexity within a class. When a class is more complex, it may contain a greater number of methods, variables, or functionalities. In such intricate classes, developers might encounter challenges related to code reuse or modularization. As a result, they may resort to code cloning as a quick solution. This also aligned with the fact that the increased cloned code size is positively correlated with a larger cloned code size as it is the third most significant explanatory factor, as shown in Table 9.

"Descending" code cloning pattern analysis

– **Code clones pairs in *"Descending"* cloned code size pattern exhibit fewer changes**. The $Model_{Descending}$ indicates a significant association between the number of changes to clone pairs and the cloned code size. When fewer modifications are made to existing pairs of code clones, it suggests that these segments of code remain relatively stable over time. This stability implies that developers are refrained from making substantial alterations or additions to the existing code clones.

[29]https://github.com/pytorch/pytorch/commit/239b38268b
[30]https://github.com/pytorch/pytorch/commit/0d8a8a2e41
[31]https://github.com/pytorch/pytorch/commit/e23827e6d6
[32]https://github.com/pytorch/pytorch/commit/05542f6222
[33]https://github.com/pytorch/pytorch/commit/812bc1dde6

– **Reduced complexity in abstract classes is associated with less reliance on code cloning**. As depicted in Table 9, a reverse association exists between the complexity of the abstract module where the code clone is originated and the cloned code size. This finding aligns with the results obtained in $Model_{Ascending}$ where an increased complexity is associated with the increased cloned code size. This suggests developers probably need to engage in less copying and pasting of code when dealing with simplified code structures in abstract classes.

"Steady" code cloning pattern analysis

– **Code optimization strategies, such as reducing the number of declaration statements and incorporating modular structures, are significantly linked to cloned code size in "Steady" cloned code patterns**. Table 5 shows an inverse association between the number of declaration statements and the stable cloned code size in $Model_{Steady}$. A lower count of declaration statements within code clones suggests a more concise and modular structure, indicating a potential effective maintenance strategy where similar functionalities across the code base share the same set of globally declared variables. This finding aligns with the relevance observed in the descending code clone pattern analysis, where an increased number of abstract classes, i.e., the second most significant factor, is also associated with stable cloned code sizes. Smaller and modular code segments are commonly linked to better maintainability and ease of understanding, highlighting the significance of cohesive code structuring practices.

Summary of RQ **2**

Within-release code cloning patterns impact the long-term code cloning trends. For instance, long-term descending trends, i.e., "Decreasing" and "Rise and Fall," in addition to "Stable" trends, consistently exhibit "Steady" within-release patterns. Within-release patterns also exhibit distinct characteristics. Our results demonstrate that *"Ascending"* code cloning pattern is associated with decreased committer involvement in clone pairs and increased cloned code size, suggesting that fewer committers may lead to a higher likelihood of cloned code size increase. In the *"Descending"* pattern, clone pairs with reduced cloned code size exhibit fewer changes, indicating stability over time, and simplified code structures in abstract classes are associated with less reliance on code cloning. Lastly, the *"Steady"* pattern links code optimization and smaller declaration statements count to stable cloned code sizes, emphasizing the significance of cohesive code structuring practices. Our work shows the characteristics of the within-release clone evolution, which helps developers prioritize their actions based on their within-release pattern.

3.4 RQ3: How do code clones manifest and evolve across different DL frameworks?

3.4.1 Motivation

Different DL frameworks support distinct features and adopt different design philosophies. For example, while *Jax* focuses on high-performance numerical computing, *PyTorch* is known for its dynamic computation graph. However, some DL frameworks share common functionalities. For example, *TensorFlow*, *PyTorch*, and *MXNet* exhibit overlapping functionalities, such as in neural network definition and training. The convergence of functionalities among these frameworks can result in comparable code patterns for similar tasks, potentially leading to code clones, as developers may employ similar constructs and operations. In this RQ, we conduct a cross-framework clone detection to identify and analyze similarities in code across different DL frameworks, and we track the evolution of the cloned functionalities. This is crucial for understanding how common functionalities of code clones manifest in DL, aiding in the development of standardized practices and fostering collaborative initiatives within the DL community.

3.4.2 Approach

The investigation process of cross-framework code clones involves the following steps:

Step 1: Building quarterly snapshots and detecting cross-framework code clones. In this step, we organize the commits of each DL framework into quarterly groups and select snapshots corresponding to the latest commit in each group for analysis. We choose the quarterly interval as it provides a fine-grained analysis of the cross-framework clone evolution. Then, we run the clone detection on the snapshots of the frameworks corresponding to the selected commits. As explained in section 2.3, we use SourcererCC to detect the cross-framework clones every quarter, generating a comprehensive list of cross-framework clone file pairs for every quarter.

Step 2: Constructing the time series for the evolution of cross-framework clones. Following the cross-framework clone detection, we construct a time series for the quarterly evolution of cross-framework clones. This time series consists of data points representing the count of cross-framework clone files on specific dates. It spans 28 quarters, offering a dynamic record of how cross-framework clones evolve across DL frameworks over time.

Step 3: Investigating cross-framework file-level clone pairs manually. To provide deeper insights into the evolution of cross-framework clones in DL frameworks, we conduct a manual analysis. In particular, we manually investigate the identified cross-framework file-level clone pairs for each snapshot. First, we examine for each file clone the *repository location*, such as the path or module, to identify the context in which the clones exist. Secondly, for every file pair, we check the *source code* to identify the functionality of these files. Lastly, we track the *code-changing activities* related to these clone files to better understand how they evolve over time and the reason they cease to be clones.

3.4.3 Results

DL frameworks present two main categories of cross-framework file-level code clones: the *functional code clones* and the *architectural adaptation clones*. *Functional code clones* include the clones where specific files, pertaining to distinct functionalities, are replicated or adapted within a DL framework. Listing 1 illustrates an example [34] of the clone from *model.py* class from the *Ray* framework belonging to this category where some code was adapted from a TensorFlow tutorial for training CNNs on the MNIST dataset to be included in the *Ray* framework. *Architectural adaptation clones* represent the adaptation and integration of an entire module from one DL framework into another resulting in an architectural adaptation between frameworks. For example, we notice that the increase in clones in March 2018 was due to the integration of the *Keras* module into *TensorFlow*. In particular, the layer and applications modules of Keras were integrated into TensorFlow as exact duplication. We also observe that even though both categories of clones are present in DL frameworks, the *architectural adaptation clones* represent the larger category among all file cross clones in DL frameworks.

```
# Most of the tensorflow code is adapted from Tensorflow's tutorial on using
# CNNs to train MNIST
# https://www.tensorflow.org/get_started/mnist/pros#build-a-multilayer-
     convolutional-network

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import ray
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import time

def download_mnist_retry(seed=0, max_num_retries=20):
    for _ in range(max_num_retries):
        try:
            return input_data.read_data_sets("MNIST_data", one_hot=True, seed=
     seed)
        except:
            pass
```

Listing 1. Example of a functional code clone instance from TensorFlow to Ray.

Functional code clones in DL frameworks belong to seven main categories *Communication Interface, Distributed Training in TensorFlow, Python Object Serialization, Efficiency in Python, Version Control, Deep Learning Architectures,* and *Probability and Statistics*. We describe below each of the categories,

[34] https://github.com/ray-project/ray/blob/6e06a9e338e1045fa0ba73b366bb78a2c7f0fef8/examples/parameter_server/model.py

and we include in Table 10 the distribution of the file clone pairs across the seven categories with the functionalities of the existing clones and the range of the lifetime of the file clone pairs.

- **Communication Interface:** Clones related to how DL frameworks handle data exchange or interaction, such as the protocol buffers (protobuf) messages and services for communication with a *BentoML* repository.
- **Distributed Training in TensorFlow:** Clones associated with distributed training in the *TensorFlow* DL framework, such as the gradient reduction (all-reduce) in distributed training using TensorFlow.
- **Python Object Serialization:** Clones related to the serialization of Python objects such as CloudPickle serialization.
- **Efficiency in Python:** Clones related to enhancing efficiency in Python, such as the LazyLoader class.
- **Version Control:** Clones pertaining to version control mechanisms, to manage and track changes in code versions such as Versioneer.
- **Deep Learning Architectures:** Clones related to deep learning architectures, suggesting shared design patterns and functionalities, such as Convolutional Neural Networks (CNN).
- **Probability and Statistics:** Clones related to probability and statistics, revealing common approaches in implementing statistical concepts, such as normal distribution.

As we notice, *Probability and Statistics* and *Deep Learning Architectures* represent the top two common categories constituting approximately 70% of the file clone pairs. This insight suggests an overlap in specific functionalities and design patterns across different DL frameworks, emphasizing commonalities in the implementation of key features such as Convolutional Neural Network (CNN) and Residual Network for the *Deep Learning Architectures* category and Normal Distribution for the *Probability and Statistics* category. We also notice that the identified DL framework pairs associated with each category reveal cross-framework code cloning instances, and 8 out of the nine studied DL frameworks present at least one cross-framework file-level clone.

The computed lifetimes of cross-framework file-level pairs vary across categories, ranging from 0 to 8 quarters, i.e., quarters. We compute the lifetimes of cross-framework file-level clone pairs to reflect how long a particular clone pair persists within a specific functional category over time. As shown in table 10, some categories, such as *Communication Interface*, experience short-lived cloning instances lasting less than a quarter, while other categories, such as *Version Control* and *Probability and Statistics* persist over a more extended period between 5 and 8 quarters. This variation suggests that code within different functional categories evolves at different rates. Some categories, such as *Version Control* and *Probability and Statistics*, may involve functionalities with more stable and fundamental design patterns less prone to frequent changes, resulting in longer lifetimes. On the other hand, categories like *Communication Interface* may experience frequent updates or changes, leading

Table 10. The distribution of functional code clones across framework.

Functional code clone categories	# of file clone pairs	Clone pair lifetime range	DL framework pairs
Communication Interface	1	0 quarter	(*BentoML*, *Paddle*)
Distributed Training in TensorFlow	1	2 quarters	(*Ray*, *TensorFlow*)
Python Object Serialization	1	1 quarter	(*Aesara*, *BentoML*)
Efficiency in Python	2	3 to 4 quarters	(*BentoML*, *TensorFlow*), (*Paddle*, *PyTorch*)
Version Control	2	8 quarters	(*Ray*, *BentoML*)
Deep Learning Architectures	5	0 to 2 quarters	(*Ray*, *TensorFlow*), (*Ray*, *PyTorch*)
Probability and Statistics	8	5 quarters	(*JAX*, *TensorFlow*), (*MXNet*, *PyTorch*)
Total	20		

to shorter lifetimes. These insights highlight areas where stability or adaptability is crucial when managing and maintaining code reuse.

Functional code clones across DL frameworks undergo a gradual disappearance, attributed to functionality evolution, code divergence, function deprecation and framework restructuring. Over time, code functionality evolves and leads to a divergence from the initial commonality. For instance, an initial commonality, such as a normal distribution, may undergo changes, as observed in *Jax*, where the file evolves to encompass additional functionalities like special functions, such as the gamma function, leading to a disappearance of the code clone across frameworks. Code divergence is another reason often resulting from adopting third-party libraries. For example, TensorFlow's shift to a new third-party library for object serialization in Python diverges from the code used in *BentoML*, which continues to use Cloudpickle. Additionally, function deprecation also leads to the decline of functional clones as frameworks undergo updates and discontinue specific functionalities. Furthermore, framework restructuring and the creation of new separate modules lead to the removal of certain functionalities. For example, the removal of Residual Network (ResNet) implementation to a standalone module outside of *PyTorch* led to the disappearance of clones between *PyTorch* and *Ray*.

***Architectural adaptation clones* between Keras and *TensorFlow* are performed gradually over multiple releases.** *TensorFlow* and *Keras* are the only two DL frameworks among the studied ones that present *architectural adaptation clones*. The evolution of cloned files between these two frameworks suggests a dynamic and evolving integration process with periods of stability, optimization, and significant updates. For instance, between March 2018 and December 2019, we observe a relatively stable number of file clone pairs, ranging from 73 to 42 pairs. This suggests an ongoing integration process where *TensorFlow* and *Keras* modules are aligning and adapting. For instance, in release *TensorFlow v1.13.1*[35], a new functionality analogous to

[35]https://github.com/tensorflow/tensorflow/releases/tag/v1.13.1

"tf.register_tensor_conversion_function" was added. Then, through the module adaptation process, these two frameworks present a decrease in the cloned file that is due to significant refactoring efforts within the *Keras* library, such as the exclusion of the two modules "applications" and "preprocessing" from Keras as described in the release *Keras 2.2.0*[36] notes. At a later stage, *TensorFlow* and *Keras* present a very small number of clones (i.e., between 0 and 5 pairs). The decrease in cloned code file pairs is attributed to optimization efforts within *TensorFlow* focusing on refining and optimizing the previously integrated *Keras* code while the development is discontinued in *Keras*, marking the full transition to the *TensorFlow* codebase.

Summary of RQ **3**

DL frameworks present *functional* and *architectural adaptation* code clones. *Functional code clones* in DL frameworks span seven categories and gradually disappear due to functionality evolution, code divergence, function deprecation, and framework restructuring. The most frequent categories, *Probability and Statistics* and *DL Architectures*, suggest commonalities in key features across different DL frameworks. This implies opportunities for collaboration, code reuse, and understanding best practices in DL framework development. Architectural adaptation clones represent the integration of one framework module into another, such as the integration of *Keras* into *TensorFlow*.

4 Implications

In this section, we discuss the possible implications of our findings that could be useful to DL Contributors within projects, DL contributors at large and Researchers for future work.

DL contributors within Projects

- **Optimizing code complexity in abstract classes**: Addressing code complexity in abstract classes is crucial to mitigating code clone proliferation, as complexity emerges as the second most explanatory factor in $Model_{Ascending}$ and $Model_{Descending}$. By emphasizing code modularization and reuse strategies, developers can potentially reduce the need for code cloning and enhance overall code maintainability. This insight underscores the importance of efficient coding practices and architectural design for DL frameworks to minimize the challenges associated with class structures. For instance, previous studies [54, 56] demonstrate that code clones can lead to a lack of good inheritance structure or abstraction.
- **Enhancing code readability and maintainability**: As demonstrated in $Model_{Stable}$, the stable cloned code size is associated with effective maintenance strategy

[36]https://github.com/keras-team/keras/releases/tag/2.2.0

in code clones, such as the reduction of the count of declaration statements and the promotion of a more concise and modular structure by using abstract classes. DL project maintainers should prioritize creating smaller, modular code components and simplify abstract class structures to enhance maintainability and ease of understanding of clone code.
- **Prioritizing testing for thick clones**: In RQ1, we observe that a significant number of bugs, i.e., more than 50% of clone-related bugs for each DL framework, are associated with "thick clones". This observation emphasizes the potential risks posed by larger and more complex clones, which highlights the importance of thorough code reviews, testing, and bug tracking in sections of code characterized by high clone thickness. Prioritizing bug detection and resolution in "thick clones" is crucial for maintaining software quality and reliability.

DL contributors at large

- **Standardizing code practices:** The results of RQ3 demonstrate that some cross-framework clones are prevalent in specific functional codes, namely Probability and Statistics and Deep Learning Architectures. These results imply a potential for shared code components in some DL common functionalities, such as architectural design, offering opportunities for promoting efficiency and consistency in development. This underscores the need to establish clear guidelines and best practices for common DL functionalities as well initiating community-driven efforts to develop and promote standardized code practices within the DL ecosystem.
- **Team size and code clone evolution:** In RQ2, our results show that the number of committers to a clone pair is a significant factor influencing cloned code size increase, as indicated by the highest χ^2 in $Model_{Ascending}$, implies that minimizing the number of developers involved in modifying a clone pair may contribute to more consistent and well-maintained code clones. This suggests that a smaller team with a better understanding of the codebase and, more specifically, the cloned code is more effective in propagating changes consistently, thus reducing the likelihood of inconsistent copies. In fact, the observation in RQ3 that there exists a consistently low community involvement in code clones complements the findings of RQ2. These insights underscore the need to encourage broader community engagement in clone-related activities and to explore strategies for code clone management and team collaboration.
- **Knowledge-sharing promotion and documentation for cloned code:** To encourage a more inclusive approach to clone maintenance, project maintainers and open-source community leaders should focus on fostering collaboration through enhanced knowledge-sharing and better documentation. Improving documentation related to cloned code can promote a wider understanding of the challenges involved, making it easier for diverse contributors to engage. By doing so, reliance on the original authors can be mitigated, and the overall sustainability and robustness of DL framework development communities can be improved.

Researchers for future work

- **Cross-Language code clones:**Research could explore code clones that span multiple platforms or programming languages. This would provide a more comprehensive understanding of code reuse and duplication in the DL ecosystem that allows researchers to assess whether similar cloning trends or patterns exist in other languages (e.g., C++, Java).
- **Impact on performance and scalability:** Researchers could investigate the potential impact of code clones on the performance and scalability of DL frameworks. This could help identify areas where code refactoring or optimization might be necessary.

5 Threats to Validity

Construct validity relates to a possible error in the data preparation. There is no established tool available to identify all existing DL frameworks. In our study, we manually curate DL frameworks from GitHub which could lead to errors. To address this potential concern, we conduct cross-validation on our selected frameworks with the most recent studies [9, 22, 26, 30, 36, 37, 71, 79, 80] on DL frameworks. Our study stands as one of the most extensive DL framework comparative investigations to date, encompassing a substantial number of Python-based DL frameworks, i.e., nine frameworks, in a single comprehensive analysis, which greatly enhances its contribution to the understanding of this field. Therefore, we assert a strong construct validity.

Another potential threat could be related to the choice of clone detection tools in our study. We employed NiCad to identify within-project code clones within each DL framework snapshot, as it has been a common choice in previous related research, such as the work by Jebnoun et al. [30] and Mo et al. [46]. The choice of detection tools might introduce certain biases or limitations. To address this, we carefully selected tools based on their precision, scalability, and relevance to our research objectives. More specifically, NiCad, a well-established tool, has been evaluated to achieve higher precision in near-miss clone (i.e., type 3) detection compared to other tools in evaluation and comparative studies of code clone detection methods [70, 72, 78]. This high precision is critical for our study, as it minimizes false positives that could otherwise skew our analysis of evolutionary trends and maintenance activities in code clones. Furthermore, NiCad supports the Python programming language, which is predominantly used in DL frameworks, and has been successfully applied in recent studies on DL-related systems [15, 86]. For cross-project code clone detection, we adopted SourcererCC, a tool that has been demonstrated to outperform other state-of-the-art tools in terms of scalability [70]. SourcererCC achieves high precision [78] while handling large codebases efficiently, as demonstrated in studies involving millions of lines of code. Given the size and complexity of the nine DL frameworks analyzed in our study, SourcererCC's scalability and high precision were crucial for managing the cross-project clone detection tasks. While newer tools, such as TACC[78], NIL[59] and Siamese[62], offer advancements in recall and extensibility, NiCad and SourcererCC have been used in a large number of prior studies. Therefore, our results on DL systems can be compared with other research that uses NiCad and SourcererCC on DL systems and non-DL systems.

Lastly, we adopt the heuristic introduced by Barbour et al. [47] to investigate the code clone bug proneness to identify bug-fix commits. This involves utilizing a predefined set of keywords associated with bug fixes. If any of these keywords are found within a commit's message, it is categorized as a bug-fix commit. This heuristic might lead to inaccurately classified commits. However, it has been successfully used in multiple previous studies from the literature [5, 30, 46, 49], and proven to achieve satisfyingly accurate results.

Internal validity relates to the concerns that might come from the internal methods used in our study. In RQ2, we offer explanations for these observed results related to practitioner development practices. However, we refrain from asserting causation and instead provide observations and correlations.

External validity relates to the potential of generalizing our study results. Given that most DL frameworks are developed in Python, our study primarily concentrates on frameworks implemented in Python. Consequently, we cannot ensure the broad applicability of our findings to frameworks developed in different programming languages, such as C++ and Java. However, we have made an effort to encompass a wide range of DL frameworks of different sizes and functionalities.

Additionally, the observed trends in code clone dynamics within DL frameworks are influenced by the unique characteristics of the DL domain, such as complex dependencies and rapidly evolving libraries [2]. Jebnoun et al. [29] find that code clones in DL possess different characteristics than clones in traditional systems, such as cloned code is likely to be more defect-prone compared to non-cloned code in DL frameworks. This observation implies that the trends identified in our study might not be directly generalized to all other types of software frameworks or programming languages. The extent to which these trends can be generalized to other types of software frameworks or libraries (e.g., web development frameworks, and game engines) would depend on the specific characteristics of those domains. However, our approach to code clone trends identification is generic and could potentially be adapted to other domains. In addition, the implications of the trends observed in DL frameworks can potentially be generalized to other domains if they suffer from similar issues. Further research is needed to explore the code clone dynamics in other types of software frameworks and other programming languages to identify any commonalities or differences.

6 Related Work

In this section, we present a review of the existing literature on analyzing DL frameworks. Additionally, we touch upon the studies relevant to code clone research.

6.1 DL framework-related empirical studies

DL, emerging as a prominent field in recent years, has attracted considerable attention within the software engineering community. A growing number of researchers have been actively empirically investigating the characteristics of DL applications and frameworks. While some researchers focused their interest on DL applications, such as identifying the challenges of building DL-based systems [55, 87] and exploring the taxonomies of faults in DL applications [22, 29], others direct their attention toward the exploration of the DL frameworks that are the building blocks of these applications.

Bug taxonomies and characteristics. Researchers contribute to understanding DL framework bugs comprehensively in several ways. TensorFlow, one of the most popular DL frameworks, has gained the attention of several researchers [31, 88]. For instance, Zhang et al. [88] analyze TensorFlow bugs sourced from StackOverflow QA pages and GitHub. The authors provide taxonomies along several dimensions, including root causes, symptoms, and bug detection strategies. Other researchers expand their work to investigate multiple DL frameworks. Chen et al. [9] conduct a comprehensive analysis of four frameworks, i.e., TensorFlow, PyTorch, MXNet, and DL4J, to identify root causes and symptoms and provide actionable guidelines for improved bug detection and debugging. In addition, the authors develop a tool, called TenFuzz, to identify bugs in Tensorflow. Similarly, Islam et al. [26] analyze the characteristics of bugs in five DL frameworks. In addition, the authors identify the stages of the DL pipeline that are more bug-prone and investigate antipatterns. Tambon et al. [73] sheds light on silent bugs in DL frameworks. The authors classify the bugs according to their impact on users' programs and the specific components where these issues originated, drawing upon information found in the issue reports. Du et al. [13] provide a classification of the fault-triggering conditions of bugs. The authors manually investigate 3,555 bug reports collected from three TensorFlow, MXNET and PaddlePaddle and analyze the frequency distribution of different bug types and their evolution features. Long et al. [42] present an exploratory study on performance and accuracy bugs in ten popular DL frameworks, revealing insights such as the primary root cause for reporting performance bugs, and they offered actionable implications for researchers, maintainers, and submitters to improve the bug management process of performance bugs. Guan et al. [18] investigate model optimization bugs (MOBs) in machine learning (ML) frameworks, focusing on bugs introduced during the optimization processes. The authors provides an empirical analysis of 371 MOBs from TensorFlow and PyTorch, examining the bugs' symptoms, root causes, detection, and fixes.

Bug fixing patterns. Jia et al. [31] explore TensorFlow, offering insights into the bugs and their fixing process within the framework's components. Ho et al. [21] replicate the study by Jia et al. to explore another popular DL framework, PyTorch. In addition to identifying the bug-fixing patterns in PyTorch, the authors provide a detailed comparison between TensorFlow and PyTorch, highlighting the similarities and differences between the frameworks' bugs. Islam et al. [27] conducted a larger scope study encompassing five DL frameworks. The authors investigate the challenges associated with 970 bug-fix patterns from Stack Overflow and GitHub and find that the most common patterns are related to data dimension and neural network connectivity issues. Li et al. [39] follow a different direction by focusing on multi-language DL frameworks. Apart from exploring the bug types and their impacts on DLF development, the authors discover that addressing bugs in multi-language frameworks involves significantly greater complexity in code changes compared to single-programming-language bug fixes.

Technical debt. Liu et al. [41] investigate the DL framework from a technical debt perspective. The authors analyze the comments indicating technical debt(self-admitted technical debt) of seven popular DL frameworks and identify seven types of technical debt in DL frameworks, i.e., design, defect, documentation, requirement, test, compatibility, and algorithm debt.

While the above work also focuses on studying DL frameworks, it only covers aspects related to bug taxonomies, bug-fixing patterns and technical debt. Our work offers an orthogonal study that investigates the evolution of clones and their impact on the maintenance of DL frameworks alongside bug-related aspects.

6.2 Code Clones

6.2.1 Code clones analysis in traditional systems

Researchers have extensively explored code clones in traditional systems [89]. These studies encompass a multifaceted exploration of code clones, including their impact on software maintenance, bug-proneness, change-proneness, and evolution.

Impact on software maintenance. Existing work demonstrates that code cloning could result in increased maintenance challenges for software systems [3, 20, 28, 51]. Mondal et al. [51] conduct a comparative empirical study to investigate the maintenance efforts required for cloned and non-cloned functions. The study found that cloned code is associated with an increased maintenance cost, in particular with Type 2 and Type 3 clones. Higo et al. [20] demonstrate that among web-based systems developed from the same specifications, those projects with a high prevalence of code clones present a greater challenge for project testing. In particular, the study demonstrates an increased effort in bug detection during unit testing and an increased number of clones during integration testing. Islam et al. [28] conduct a comparative analysis of code clones with and without bugs, considering 29 code quality metrics across 2,077 revisions of three Java software systems and offering insights for cost-effective clone management and clone-aware software development.

Bug proneness. Several previous studies have examined the bug proneness of code clones, revealing that clones make code more bug-prone and increase maintenance costs [24, 25, 32, 63, 76]. Islam et al. [25] compare the bug-proneness of clone and non-clone code. This paper presents a comparative study on the bug proneness of code clones and non-clones, finding that clone code has a significantly higher percentage of files changed due to bug-fix commits and a greater likelihood of severe bugs, suggesting that bug-fixing changes in clone code require extra attention. In another study, Islam et al. [24] delve into the bug-proneness of micro-clones, i.e., small code fragments of 1 to 4 lines of code and compare them with regular code clones in diverse open-source systems written in C, C#, and Java. The study findings show that micro-clones are significantly more prone to bugs, exhibit more consistent changes due to bug-fix commits, affect a higher percentage of files, and contain a greater percentage of severe bugs than regular clones, hence underscoring the importance of managing and maintaining micro-clones in software development. Jiang et al. [32] study also validates that code clones had a higher likelihood of containing bugs, primarily originating from inconsistencies within cloned code segments.

Change proneness. The negative impacts of code clones on software maintenance have led to extensive research on clone stability and change proneness. [43, 44, 48, 52, 53, 57]. For example, Mostafa [57] focuses on analyzing clone evolution with respect to clone location, i.e., Inter-File and Intra-File, and clone lifetime. The study reveals that Intra-File clones are more prevalent, suggesting that developers tend to duplicate code within the same file, and these clones are also more dynamic, indicating a preference for refactoring or altering clones within the same file. Mondal et al. [48] find that cloned code tends to be more change-prone and unstable during the maintenance phase than non-cloned code. The study also identifies differences in stability among various types of clones, programming languages, and programming paradigms. In a more recent study, Mondal et al. [53] conduct a comprehensive study on 12 subject systems to compare the stability of clone and non-clone codes. The authors demonstrate that code clones generally exhibit higher change-proneness as compared to non-clones by referring to the eight stability measuring metrics, implying increased maintenance effort and cost.

Clone evolution. Some researchers have constructed clone genealogies by tracing the history of code clones [4, 74, 75]. Barbour et al .[4] derive six distinct evolutionary patterns by building the clone genealogies of four open-source Java systems. In addition, the authors leverage the clone genealogy information to enhance the effectiveness of fault prediction models. Similarly, Thongtanunam et al. [74] conduct an empirical study on six open-source Java systems genealogies, revealing that a significant proportion of clones have short lifespans. In addition, the authors predict, using random forest classifiers, whether a newly introduced clone would be short-lived based on factors extracted from the genealogy. In a recent work, Bladel and Demeyer [75] conduct a study on eight open-source systems genealogies and revealed that code clones are more prevalent in test code compared to production code due to the recurring pattern of unit test code.

The studies mentioned above highlight the challenges of code cloning in traditional software on software maintenance, including bug proneness, change proneness and the clone evolution. However, given the differences between traditional and DL frameworks, we cannot assume those findings directly apply. Our work aims to address the knowledge gap concerning the evolution of code cloning and its implications within the DL frameworks.

6.2.2 Code clones analysis in DL systems

Despite the increasing surge in the development of DL software, only two studies have investigated code clones within this domain, highlighting the need for more comprehensive investigations into code cloning practices in DL software.

Jebnoun et al. [30] introduce the first study on clones in DL development. The authors analyze code clones in 59 Python, 14 C#, and 6 Java-based DL systems and an equal number of traditional software, highlighting the frequency, distribution, and effects of code clones. In addition, they provide a taxonomy to identify phases with a higher risk of cloning-related faults. Their findings indicate that code cloning is prevalent in DL systems, with developers often copying code from files located in other directories, and that code cloning is more common during DL model creation, training, and data preprocessing phases. In addition, the authors find that code clones in DL code are likely to be more defect-prone compared to non-cloned code.

Similarly, Mo et al. [46] also find that code clones are prevalent in DL systems, exhibiting nearly twice the rate in traditional projects. However, the authors conduct their study on a dataset of Python projects with only 45 DL and 45 traditional projects. Different from the work of Jebnoun et al., Mo et al. focus on co-changed clones and investigate the distribution of Type 1, Type 2, and Type 3 co-changes and their bug-proneness. The authors also conduct a comparative analysis of the DL applications subject based on the underlying DL frameworks.

This paper conducts an empirical study of code clone evolution in DL frameworks. Our work differs from the aforementioned previous work in two ways: firstly, we study code clones in DL frameworks rather than the DL applications, and secondly, our analysis focuses on the evolutionary and comparative aspects of code clones as opposed to merely studying the distribution of clones within a single snapshot. Moreover, we investigate the code clones across DL frameworks and not only within a framework. We also examine the clone trend within releases and provide developers with insights for stable, maintainable clone practices.

7 Conclusion

Given the pivotal role of DL frameworks in the rapidly growing field of artificial intelligence and machine learning, its software quality is crucial for the development of DL-based applications. Code clone is a common coding practice that can potentially adversely affect software maintainability. In this paper, we conduct an empirical analysis of nine popular DL frameworks, including TensorFlow, Paddle, PyTorch, Aesara, Ray, MXNet, Keras, Jax, and BentoML, to reveal insights into the long-term code clone trend over releases, and within-release clone patterns, i.e., short-term, and their implications. In addition, we also conduct a file-level cross-framework clones analysis to identify the cross-functionalities of cloned code within the frameworks.

Our results identify four distinct trends in code clone evolution. In addition, we provide an understanding of the factors influencing cloned code size reduction and unveil the association between within-release clone patterns and long-term clone trends. Our research contributes to the foundational understanding of code cloning in the context of DL frameworks. The identification of cross-framework file-level code clones, categorized as *functional* and *architectural adaptation* code clones, highlights collaborative opportunities and commonalities in key features across different DL frameworks. Moreover, our study offers insights emphasizing the importance of fostering collaboration, simplifying abstract class complexity, optimizing code maintainability, and standardizing code practices within DL frameworks. As the DL landscape continues to evolve, our findings provide valuable insights to the DL community, contributing to the sustainable development and maintenance of these critical systems.

In our future work, we plan to broaden the scope of our study beyond Python projects to include other programming languages, such as C++ and Java, aiming for a more comprehensive understanding of code cloning in diverse DL framework language ecosystems. By exploring the trends and clone dynamics in DL frameworks written in these programming languages, we can identify any language-specific factors that influence code reuse and duplication. Additionally, we aim to investigate whether the observed trends in DL frameworks can be generalized to other types of software frameworks, which might provide valuable insights into the broader implications of code cloning trends and their impact on software development practices.

Acknowledgments

The first author would like to acknowledge the support of NSERC Vanier Canada Graduate Scholarships.

References

[1] 2024. Github Ranking. https://github.com/EvanLi/Github-Ranking#most\protect\discretionary{\char\hyphenchar\font}{}{}stars.

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[3] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. 2007. How Clones are Maintained: An Empirical Study. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. 81–90. https://doi.org/10.1109/CSMR.2007.26

[4] Liliane Barbour, Le An, Foutse Khomh, Ying Zou, and Shaohua Wang. 2018. An investigation of the fault-proneness of clone evolutionary patterns. *Software Quality Journal* 26 (2018), 1187–1222.

[5] Liliane Barbour, Foutse Khomh, and Ying Zou. 2011. Late propagation in software clones. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 273–282. https://doi.org/10.1109/ICSM.2011.6080794

[6] Liliane Barbour, Foutse Khomh, and Ying Zou. 2013. An empirical study of faults in late propagation clone genealogies. *Journal of Software: Evolution and Process* 25 (11 2013). https://doi.org/10.1002/smr.

1597

[7] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. 2018. The Open-Closed Principle of Modern Machine Learning Frameworks. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 353–363.

[8] Donald J. Berndt and James Clifford. 1994. Using Dynamic Time Warping to Find Patterns in Time Series. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining* (Seattle, WA) *(AAAIWS'94)*. AAAI Press, 359–370.

[9] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward Understanding Deep Learning Framework Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 135 (sep 2023), 31 pages. https://doi.org/10.1145/3587155

[10] Muslim Chochlov, Gul Aftab Ahmed, James Vincent Patten, Guoxian Lu, Wei Hou, David Gregg, and Jim Buckley. 2022. Using a Nearest-Neighbour, BERT-Based Approach for Scalable Clone Detection. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 582–591. https://doi.org/10.1109/ICSME55016.2022.00080

[11] J. Cohen, P. Cohen, S.G. West, and L.S. Aiken. 2002. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences* (3rd ed.). Routledge. https://doi.org/10.4324/9780203774441

[12] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. 2017. Scripted GUI Testing of Android Apps: A Study on Diffusion, Evolution and Fragility. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering* (Toronto, Canada) *(PROMISE)*. Association for Computing Machinery, New York, NY, USA, 22–32. https://doi.org/10.1145/3127005.3127008

[13] Xiaoting Du, Yulei Sui, Zhihao Liu, and Jun Ai. 2023. An Empirical Study of Fault Triggers in Deep Learning Frameworks. *IEEE Transactions on Dependable and Secure Computing* 20, 4 (2023), 2696–2712. https://doi.org/10.1109/TDSC.2022.3152239

[14] Osama Ehsan, Foutse Khomh, Ying Zou, and Dong Qiu. 2023. Ranking Code Clones to Support Maintenance Activities. *Empirical Softw. Engg.* 28, 3 (apr 2023), 38 pages. https://doi.org/10.1007/s10664-023-10292-0

[15] Siyue Feng, Wenqi Suo, Yueming Wu, Deqing Zou, Yang Liu, and Hai Jin. 2024. Machine Learning is All You Need: A Simple Token-based Approach for Effective Code Clone Detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 222, 13 pages. https://doi.org/10.1145/3597503.3639114

[16] Y. Golubev and T. Bryksin. 2021. On the Nature of Code Cloning in Open-Source Java Projects. In *2021 IEEE 15th International Workshop on Software Clones (IWSC)*. IEEE Computer Society, Los Alamitos, CA, USA, 22–28. https://doi.org/10.1109/IWSC53727.2021.00010

[17] Florin Gorunescu. 2011. *Data Mining: Concepts, Models and Techniques.*

[18] Hao Guan, Ying Xiao, Jiaying Li, Yepang Liu, and Guangdong Bai. 2023. A Comprehensive Study of Real-World Bugs in Machine Learning Model Optimization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 147–158. https://doi.org/10.1109/ICSE48619.2023.00024

[19] James A. Hanley and Barbara J. McNeil. 1982. The Meaning and Use of the Area Under a Receiver Operating Characteristic (ROC) Curve. *Radiology* 143, 1 (1982), 29–36. https://doi.org/10.1148/radiology.143.1.7063747

[20] Yoshiki Higo, Shinsuke Matsumoto, Shinji Kusumoto, Takashi Fujinami, and Takashi Hoshino. 2018. Correlation analysis between code clone metrics and project data on the same specification projects. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*. 37–43. https://doi.org/10.1109/IWSC.2018.8327317

[21] Sharon Chee Yin Ho, Vahid Majdinasab, Mohayeminul Islam, Diego Elias Costa, Emad Shihab, Foutse Khomh, Sarah Nadi, and Muhammad Raza. 2023. An Empirical Study on Bugs Inside PyTorch: A Replication Study. arXiv:2307.13777 [cs.SE]

[22] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. https://doi.org/10.1145/3377811.3380395

[23] Katsuro Inoue. 2021. *Introduction to Code Clone Analysis*. Springer Singapore, Singapore, 3–27. https://doi.org/10.1007/978-981-16-1927-4_1

[24] Judith F. Islam, Manishankar Mondal, and Chanchal K. Roy. 2019. A Comparative Study of Software Bugs in Micro-clones and Regular Code Clones. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 73–83. https://doi.org/10.1109/SANER.2019.8667993

[25] Judith F. Islam, Manishankar Mondal, Chanchal Kumar Roy, and Kevin A. Schneider. 2017. A Comparative Study of Software Bugs in Clone and Non-Clone Code. In *International Conference on Software Engineering and Knowledge Engineering*. https://api.semanticscholar.org/CorpusID:34980604

[26] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[27] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks: Fix Patterns and Challenges. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1135–1146. https://doi.org/10.1145/3377811.3380378

[28] Md Rakibul Islam and Minhaz F. Zibran. 2018. On the characteristics of buggy code clones: A code quality perspective. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*. 23–29. https://doi.org/10.1109/IWSC.2018.8327315

[29] Hadhemi Jebnoun, Houssem Ben Braiek, Mohammad Masudur Rahman, and Foutse Khomh. 2020. The Scent of Deep Learning Code: An Empirical Study. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 420–430. https://doi.org/10.1145/3379597.3387479

[30] Hadhemi Jebnoun, Md Saidur Rahman, Foutse Khomh, and Biruk Asmare Muse. 2022. Clones in Deep Learning Code: What, Where, and Why? *Empirical Softw. Engg.* 27, 4 (jul 2022), 75 pages. https://doi.org/10.1007/s10664-021-10099-x

[31] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2021. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software* 177 (2021), 110935. https://doi.org/10.1016/j.jss.2021.110935

[32] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-Based Detection of Clone-Related Bugs *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 55–64. https://doi.org/10.1145/1287624.1287634

[33] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Christoph Treude. 2020. The Impact of Automated Feature Selection Techniques on the Interpretation of Defect Models. *Empirical Softw. Engg.* 25, 5 (sep 2020), 3590–3638. https://doi.org/10.1007/s10664-020-09848-1

[34] Cory Kapser and Michael W. Godfrey. 2006. "Cloning Considered Harmful" Considered Harmful. In *2006 13th Working Conference on Reverse Engineering*. 19–28. https://doi.org/10.1109/WCRE.2006.1

[35] Taghi M. Khoshgoftaar and Edward B. Allen. 1999. LOGISTIC REGRESSION MODELING OF SOFTWARE QUALITY. *International Journal of Reliability, Quality and Safety Engineering* 06 (1999), 303–317. https://api.semanticscholar.org/CorpusID:61246258

[36] Misoo Kim, Youngkyoung Kim, and Eunseok Lee. 2021. Denchmark: A Bug Benchmark of Deep Learning-related Software. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 540–544. https://doi.org/10.1109/MSR52588.2021.00070

[37] Max Langenkamp and Daniel N. Yue. 2022. How Open Source Machine Learning Software Shapes AI. In *Proceedings of the 2022 AAAI/ACM Conference on AI, Ethics, and Society* (Oxford, United Kingdom) *(AIES '22)*. Association for Computing Machinery, New York, NY, USA, 385–395. https://doi.org/10.1145/3514094.3534167

[38] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. 2006. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* 32, 3 (2006), 176–192. https://doi.org/10.1109/TSE.2006.28

[39] Zengyang Li, Sicheng Wang, Wenshuo Wang, Peng Liang, Ran Mo, and Bing Li. 2023. Understanding Bugs in Multi-Language Deep Learning Frameworks. arXiv:2303.02695 [cs.SE]

[40] Charles X. Ling, Jin Huang, and Harry Zhang. 2003. AUC: A Better Measure than Accuracy in Comparing Learning Algorithms. In *Canadian Conference on AI*. https://api.semanticscholar.org/CorpusID:18614857

[41] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2021. An Exploratory Study on the Introduction and Removal of Different Types of Technical Debt in Deep Learning Frameworks. *Empirical Softw. Engg.* 26, 2 (mar 2021), 36 pages. https://doi.org/10.1007/s10664-020-09917-5

[42] Guoming Long and Tao Chen. 2022. On Reporting Performance and Accuracy Bugs for Deep Learning Frameworks: An Exploratory Study from GitHub *(EASE '22)*. Association for Computing Machinery, New York, NY, USA, 90–99. https://doi.org/10.1145/3530019.3530029

[43] Angela Lozano and Michel Wermelinger. 2008. Assessing the effect of clones on changeability. In *2008 IEEE International Conference on Software Maintenance*. 227–236. https://doi.org/10.1109/ICSM.2008.4658071

[44] Angela Lozano and Michel Wermelinger. 2010. Tracking Clones' Imprint. In *Proceedings of the 4th International Workshop on Software Clones* (Cape Town, South Africa) *(IWSC '10)*. Association for Computing Machinery, New York, NY, USA, 65–72. https://doi.org/10.1145/1808901.1808910

[45] Shane Mcintosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2016. An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Softw. Engg.* 21, 5 (oct 2016), 2146–2189. https://doi.org/10.1007/s10664-015-9381-9

[46] Ran Mo, Yao Zhang, Yushuo Wang, Siyuan Zhang, Pu Xiong, Zengyang Li, and Yang Zhao. 2023. Exploring the Impact of Code Clones on Deep Learning Software. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 153 (sep 2023), 34 pages. https://doi.org/10.1145/3607181

[47] Mockus and Votta. 2000. Identifying reasons for software changes using historic databases. In *Proceedings 2000 International Conference on Software Maintenance*. 120–130. https://doi.org/10.1109/ICSM.2000.883028

[48] Manishankar Mondal, Md Saidur Rahman, Chanchal K. Roy, and Kevin A. Schneider. 2018. Is Cloned Code Really Stable? *Empirical Softw. Engg.* 23, 2 (apr 2018), 693–770. https://doi.org/10.1007/s10664-017-9528-y

[49] Manishankar Mondal, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. Investigating Context Adaptation Bugs in Code Clones. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 157–168. https://doi.org/10.1109/ICSME.2019.00026

[50] Manishankar Mondal, Chanchal Roy, Md Saidur Rahman, Ripon Saha, Jens Krinke, and Kevin Schneider. 2012. Comparative Stability of Cloned and Non-cloned Code: An Empirical Study. *Proceedings of the ACM Symposium on Applied Computing*, 1227–1234. https://doi.org/10.1145/2245276.2231969

[51] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. 2017. Does cloned code increase maintenance effort?. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)*. 1–7. https://doi.org/10.1109/IWSC.2017.7880507

[52] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. 2020. A fine-grained analysis on the inconsistent changes in code clones. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 220–231.

[53] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. 2021. *A Summary on the Stability of Code Clones and Current Research Trends.* Springer Singapore, Singapore, 169–180. https://doi.org/10.1007/978-981-16-1927-4_12

[54] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. 2002. Software quality analysis by code clones in industrial legacy software. In *Proceedings Eighth IEEE Symposium on Software Metrics*. 87–94. https://doi.org/10.1109/METRIC.2002.1011328

[55] Mohammad Mehdi Morovati, Florian Tambon, Mina Taraghi, Amin Nikanjam, and Foutse Khomh. 2023. Common Challenges of Deep Reinforcement Learning Applications Development: An Empirical Study. arXiv:2310.09575 [cs.SE]

[56] M. Monzur Morshed, Md Rahman, and Salah Ahmed. 2012. A Literature Review of Code Clone Analysis to Improve Software Maintenance Process. (05 2012).

[57] Md. Jubair Ibna Mostafa. 2019. An Empirical Study on Clone Evolution by Analyzing Clone Lifetime. In *2019 IEEE 13th International Workshop on Software Clones (IWSC)*. 20–26. https://doi.org/10.1109/

IWSC.2019.8665850

[58] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. 2008. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 521–530. https://doi.org/10.1145/1368088.1368160

[59] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. 2021. NIL: large-scale detection of large-variance clones. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 830–841. https://doi.org/10.1145/3468264.3468564

[60] Shayan Noei, Heng Li, Stefanos Georgiou, and Ying Zou. 2023. An Empirical Study of Refactoring Rhythms and Tactics in the Software Development Process. *IEEE Transactions on Software Engineering* 49, 12 (2023), 5103–5119. https://doi.org/10.1109/TSE.2023.3326775

[61] P. Pinheiro. 2010. *Linear and Nonlinear Mixed Effects Models: Theory and Applications*. https://cran.r-project.org/web/packages/nlme/nlme.pdf

[62] Chaiyong Ragkhitwetsagul and Jens Krinke. 2019. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Softw. Engg.* 24, 4 (Aug. 2019), 2236–2284. https://doi.org/10.1007/s10664-019-09697-7

[63] Md Saidur Rahman and Chanchal K. Roy. 2017. On the Relationships Between Stability and Bug-Proneness of Code Clones: An Empirical Study. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 131–140. https://doi.org/10.1109/SCAM.2017.26

[64] W. Rahman, Y. Xu, F. Pu, J. Xuan, X. Jia, M. Basios, L. Kanthan, L. Li, F. Wu, and B. Xu. 2020. Clone Detection on Large Scala Codebases. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*. IEEE Computer Society, Los Alamitos, CA, USA, 38–44. https://doi.org/10.1109/IWSC50091.2020.9047640

[65] Stephen Romansky, Cheng Chen, Baljeet Malhotra, and Abram Hindle. 2018. Sourcerer's Apprentice and the study of code snippet migration. *CoRR* abs/1808.00106 (2018). arXiv:1808.00106 http://arxiv.org/abs/1808.00106

[66] Peter J. Rousseeuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.* 20 (1987), 53–65. https://doi.org/10.1016/0377-0427(87)90125-7

[67] Chanchal Roy and James Cordy. 2007. A Survey on Software Clone Detection Research. *School of Computing TR 2007-541* (01 2007).

[68] Chanchal K. Roy and James R. Cordy. 2008. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *2008 16th IEEE International Conference on Program Comprehension*. 172–181. https://doi.org/10.1109/ICPC.2008.41

[69] Chanchal K. Roy and James R. Cordy. 2009. A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*. 157–166. https://doi.org/10.1109/ICSTW.2009.18

[70] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877

[71] Ali Shatnawi, Ghadeer Al-Bdour, Raffi Al-Qurran, and Mahmoud Al-Ayyoub. 2018. A comparative study of open source deep learning frameworks. In *2018 9th International Conference on Information and Communication Systems (ICICS)*. 72–77. https://doi.org/10.1109/IACS.2018.8355444

[72] Jeffrey Svajlenko and Chanchal K. Roy. 2014. Evaluating Modern Clone Detection Tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 321–330. https://doi.org/10.1109/ICSME.2014.54

[73] Florian Tambon, Amin Nikanjam, Le An, Foutse Khomh, and Giuliano Antoniol. 2021. Silent Bugs in Deep Learning Frameworks: An Empirical Study of Keras and TensorFlow. *ArXiv* abs/2112.13314 (2021). https://api.semanticscholar.org/CorpusID:245502137

[74] Patanamon Thongtanunam, Weiyi Shang, and Ahmed E. Hassan. 2019. Will This Clone Be Short-Lived? Towards a Better Understanding of the Characteristics of Short-Lived Clones. *Empirical Softw. Engg.* 24, 2 (apr 2019), 937–972. https://doi.org/10.1007/s10664-018-9645-2

[75] Brent van Bladel and Serge Demeyer. 2023. A Comparative Study of Code Clone Genealogies in Test Code and Production Code. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 913–920. https://doi.org/10.1109/SANER56733.2023.00110

[76] Stefan Wagner, Asim Abdulkhaleq, Kamer Kaya, and Alexander Paar. 2016. On the Relationship of Inconsistent Software Clones and Faults: An Empirical Study. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 79–89. https://doi.org/10.1109/SANER.2016.94

[77] Zhiyuan Wan, Xin Xia, David Lo, and Gail C. Murphy. 2021. How does Machine Learning Change Software Development Practices? *IEEE Transactions on Software Engineering* 47, 9 (2021), 1857–1871. https://doi.org/10.1109/TSE.2019.2937083

[78] Yuekun Wang, Yuhang Ye, Yueming Wu, Weiwei Zhang, Yinxing Xue, and Yang Liu. 2023. Comparison and Evaluation of Clone Detection Techniques with Different Code Representations. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 332–344. https://doi.org/10.1109/ICSE48619.2023.00039

[79] Zhaobin Wang, Ke Liu, Jian Li, Ying Zhu, and Yaonan Zhang. 2019. Various Frameworks and Libraries of Machine Learning and Deep Learning: A Survey. *Archives of Coputational Methods in Engineering* (02 2019). https://doi.org/10.1007/s11831-018-09312-w

[80] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep Learning Library Testing via Effective Model Generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 788–799. https://doi.org/10.1145/3368089.3409761

[81] T. Warren Liao. 2005. Clustering of time series data—a survey. *Pattern Recognition* 38, 11 (2005), 1857–1874. https://doi.org/10.1016/j.patcog.2005.01.025

[82] S. Weisberg. 2005. *Applied Linear Regression*. Vol. 528. John Wiley & Sons.

[83] Aidan Z. H. Yang, Safwat Hassan, Ying Zou, and A. Hassan. 2022. An empirical study on release notes patterns of popular apps in the Google Play Store. *Empirical Software Engineering* 27 (2022), 1–38. https://api.semanticscholar.org/CorpusID:238863459

[84] Ruru Yue, Zhe Gao, Na Meng, Yingfei Xiong, Xiaoyin Wang, and J. David Morgenthaler. 2018. Automatic Clone Recommendation for Refactoring Based on the Present and the Past. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 115–126. https://doi.org/10.1109/ICSME.2018.00021

[85] Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. 2023. A systematic literature review on source code similarity measurement and clone detection: techniques, applications, and challenges. arXiv:2306.16171 [cs.SE]

[86] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086

[87] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 104–115. https://doi.org/10.1109/ISSRE.2019.00020

[88] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866

[89] Yan Zhong, Xunhui Zhang, Wang Tao, and Yanzhi Zhang. 2022. A Systematic Literature Review of Clone Evolution. In *Proceedings of the 5th International Conference on Computer Science and Software Engineering* (Guilin, China) *(CSSE '22)*. Association for Computing Machinery, New York, NY, USA,

461–473. https://doi.org/10.1145/3569966.3570091