Demystifying the Evolution of Neural Networks with BOM Analysis: Insights from a Large-Scale Study of 55,997 GitHub Repositories

Xiaoning Ren[1], Yuhang Ye[1], Xiongfei Wu[2], Yueming Wu[3], Yinxing Xue* [4]
[1] University of Science and Technology of China, China, hnurxn@mail.ustc.edu.cn, yyh834771838@mail.ustc.edu.cn
[2] University of Luxembourg, Luxembourg, xiongfei.wu@uni.lu
[3] Huazhong University of Science and Technology, China , wuyueming21@gmail.com
[4] Institute of AI for Industries, China, yxxue@iaii.ac.cn

***Abstract*—Neural networks have become integral to many fields due to their exceptional performance. The open-source community has witnessed a rapid influx of neural network (NN) repositories with fast-paced iterations, making it crucial for practitioners to analyze their evolution to guide development and stay ahead of trends. While extensive research has explored traditional software evolution using Software Bill of Materials (SBOMs), these are ill-suited for NN software, which relies on pre-defined modules and pre-trained models (PTMs) with distinct component structures and reuse patterns. Conceptual AI Bills of Materials (AIBOMs) also lack practical implementations for large-scale evolutionary analysis. To fill this gap, we introduce the Neural Network Bill of Material (NNBOM), a comprehensive dataset construct tailored for NN software. We create a large-scale NNBOM database from 55,997 curated PyTorch GitHub repositories, cataloging their TPLs, PTMs, and modules. Leveraging this database, we conduct a comprehensive empirical study of neural network software evolution across software scale, component reuse, and inter-domain dependency, providing maintainers and developers with a holistic view of its long-term trends. Building on these findings, we develop two prototype applications, *Multi repository Evolution Analyzer* and *Single repository Component Assessor and Recommender*, to demonstrate the practical value of our analysis.**

***Index Terms*—Neural Network, Software Evolution Analysis, Bill of Material**

I. Introduction

Neural networks have become indispensable in modern AI due to their ability to model complex patterns and learn from large datasets. Their usage has rapidly expanded across diverse domains, including natural language processing, computer vision, smart healthcare [1], and autonomous driving [2]. This widespread adoption has led to a surge in neural network repositories, posing significant challenges for maintenance. Therefore, conducting large-scale evolution analysis and gaining a deeper understanding of the broader ecosystem's evolution are essential. Such analysis provides valuable insights into emerging trends and future directions, enabling practitioners to identify best development practices, enhance development efficiency, make informed decisions when selecting neural network technologies, and help community maintainers monitor the ecosystem's status and optimize maintenance strategies.

Existing research on software evolution has provided valuable insights into the development of traditional software systems, primarily through the analysis of software components and dependency relationships, which are often documented in a Software Bill of Materials (SBOM). From the perspective of software components, analyzing evolution helps track the introduction, modification, and removal of libraries, modules, and functionalities, providing insights into how systems improve in complexity and efficiency [3]–[7]. Regarding dependency relationships, evolution analysis uncovers how software systems become increasingly interconnected, with modules and external libraries relying on each other [8]–[11]. This not only aids in optimizing the architecture but also highlights potential bottlenecks or vulnerabilities in the dependency chain, ensuring smoother maintenance and upgrades. However, there is a notable lack of comprehensive evolution analysis specifically targeting neural network repositories.

Neural network software differs significantly from traditional software in terms of components and dependencies. Traditional software often involves detailed logical flows, including numerous conditional and loop statements, and typically relies on a wide range of third-party libraries (TPLs) and frameworks, making its dependency management complex. The relationships between these components are deeply intertwined, requiring careful attention to compatibility, updates, and potential conflicts. In contrast, neural networks are primarily composed of modules, such as predefined layers (e.g., convolutional or pooling layers) and pre-trained models (PTMs), which are reused across various projects. This results in simpler dependencies compared to the intricate dependency structures found in traditional software. Consequently, existing SBOMs designed for traditional software are ill-suited for analyzing the evolution of neural network software. Meanwhile, discussions around the AI Bill of Materials (AIBOM) have largely remained conceptual, focusing on what kinds of information should be included to promote transparency, but lacking concrete implementations or empirical validation [12]–[14].

* Corresponding author. Xiaoning Ren and Yuhang Ye contributed equally.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00170

To address this gap, we define a practically applicable Bill of Materials tailored for neural networks, termed the Neural Network Bill of Materials (NNBOM). Based on this definition, we construct a large-scale dataset from the open-source community to support reliable neural network evolution analysis. Specifically, we design a systematic process for building the NNBOM database. This process includes data collection from GitHub, extraction and analysis of key neural network components—such as third-party libraries (TPLs), pre-trained models (PTMs), and custom modules—and subsequent data processing to compile the final dataset. The result is a comprehensive database of 55,997 high-quality PyTorch repositories.

Using this database, we conduct a comprehensive analysis of neural network software evolution, revealing several key insights: ①**Software Scale Evolution**: The growing module count alongside stable module size suggests a prevailing convention of managing complexity through finer-grained modularization. ②**Component Reuse Evolution**: All components show increasing integration into larger co-usage communities, with PTMs and Modules further driving functional diversification. ③**Inter-Domain Dependency Evolution**: Rising cross-domain reuse reflects a shift toward general-purpose, longer-lived modules, making domain adaptability a quantifiable proxy for module quality and stability. Finally, we develop two prototype tools: *Multi-repository Evolution Analyzer*, which tracks component evolution triggered by newly added repositories over a given period; and *Single-repository Component Assessor and Recommender*, which analyzes the component structure of an individual repository and recommends relevant components and similar repositories. In summary, the main contributions of this paper are as follows:

- **The First BOM-Based Evolution Analysis of Neural Network Software**: To the best of our knowledge, this is the first work to define and construct a bill of materials tailored for neural networks (NNBOM) and to conduct an evolution analysis based on it.
- **A Comprehensive and Open Dataset**: We build and release a large-scale dataset comprising 55,997 curated high-quality PyTorch repositories [15], offering a solid foundation for future research.
- **Multi-perspective Evaluation**: We provide an in-depth analysis of neural network software evolution from three key perspectives: software scale, component reuse, and inter-domain dependency. This approach yields valuable insights for both researchers and practitioners.

II. Background and Related Work

A. Terminology

Our study focuses on GitHub projects, primarily considering five software artifacts. Figure 1 illustrates the key concepts in this study.

①**Repository.** A repository is a storage location for organizing and managing project code, files, and their related version histories.

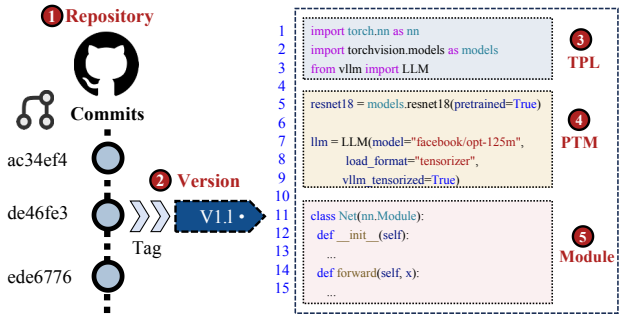

Fig. 1: Key concepts in this research.

②**Version.** A version in GitHub is a specific repository state marked by tagging a point in the commit history. These tags represent significant milestones or releases, allowing us to track and analyze the repository's development over time.

③**Third-Party Library.** A TPL is a pre-written collection of code and resources developed by an external entity, which developers can integrate into their projects using an `import` instruction.

④**Pre-Trained Model.** A PTM is a neural network model previously trained on a large dataset and can be used directly through function invocations.

⑤**Module.** In the PyTorch framework, a module refers to any class that inherits from or indirectly derives from "torch.nn.Module". These modules serve as fundamental building blocks for constructing neural network architectures, encapsulating layers, and other components necessary for defining and training models.

⑥**SBOM.** A Software Bill of Materials (SBOM), typically designed for traditional software, is a comprehensive inventory detailing all libraries, frameworks, modules, and other third-party components used in software development.

⑦**NNBOM.** In contrast, NNBOM is a bill of materials specifically designed to account for the unique characteristics of neural networks, such as simpler dependencies and greater code reuse, and includes components ③ to ⑤.

B. Related Work

Software Evolution. The study of software clone evolution provides key insights into how software systems develop over time. Kim *et al.* [3] first analyzed software clones' evolution, examining clone fragments across versions. Juergens *et al.* [4] conducted a large-scale case study on five software systems, revealing how different clone evolution patterns impact maintenance. Barbour *et al.* [5], and Thongtanunam *et al.* [6] advanced this research by constructing clone genealogies to analyze evolution patterns. ASSI *et al.* [7] extended clone evolution studies to deep learning frameworks, analyzing trends in clone code size, bug-proneness, and community size. Beyond single projects, research has examined software evolution at a community level by analyzing inter-project clones and dependencies. Kikas *et al.* [8] analyzed dependency network evolution in the JavaScript, Ruby, and Rust ecosystems.

Wittern *et al.* [9] studied the npm ecosystem by analyzing metadata and historical changes in package dependencies, revealing the dynamic nature of dependency relationships. Decan *et al.* [10] compared dependency network evolution across seven software packaging ecosystems, highlighting commonalities and differences in dependency dynamics. Yang *et al.* [11] reviewed major language models from 2018 to 2023, analyzing the rise and fall of different approaches to provide a clear understanding of large language models' development. However, there is currently a lack of large-scale evolution analysis specifically targeting neural network software.

SBOM. SBOMs provide comprehensive transparency into the components used in traditional software systems. Carmody et al. [16] demonstrated how such transparency enhances the trustworthiness, resilience, and security of medical software, benefiting software producers, consumers, and regulators. Although various tools have been developed for SBOM generation [17], they are generally ineffective for neural network software due to fundamental differences, due to their model-driven architecture, and the lack of static, inspectable dependencies. The concept of the AI Bill of Materials (AIBOM) was introduced by Chan *et al.* in 2017 [12]. Recent work has explored how AIBOM differs from traditional SBOM [14], and what kinds of information should be included to promote transparency and accountability in AI development [18]–[20]. However, these discussions have largely remained theoretical, with limited practical implementation. Given these limitations, we decided to develop NNBOM, a dataset specifically designed for neural network software, which will serve as the basis for neural network software evolution analysis.

C. Motivation

While traditional software evolution is well-studied, there exists a significant gap in understanding the unique evolutionary dynamics of NN software. To date, there has been no large-scale, component-centric evolution analysis tailored to NN systems. Existing studies often focus on traditional software, with its complex logic and dependency management, making their methods, which often rely on conventional SBOMs, less applicable to the unique characteristics of NN software. NN software is distinctly characterized by the heavy reuse of specialized components like pre-defined architectural modules and PTMs, which have different reuse patterns. While conceptual AIBOMs aim for broader AI transparency, they are not yet practical tools for systematically tracking the evolution of these core NN software artifacts at scale.

This work is motivated by the need for a practical framework to construct a Bill of Materials specifically for neural network software, enabling empirical analysis of its component-level evolution. To this end, we introduce NNBOM, a dataset that catalogs key NN software artifacts, namely third-party libraries (TPLs), pretrained models (PTMs), and custom modules. Neural network implementations are fundamentally constructed from these three types of building blocks: PTMs that are adopted for reuse, custom modules developed by practitioners, and TPLs such as PyTorch that provide the essential infrastructure.

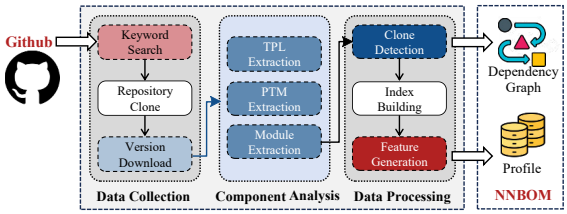

Fig. 2: Overview of NNBOM database construction.

By systematically extracting these components, NNBOM holistically captures how NN software is built and how it evolves over time. Distinct from SBOMs/AIBOMs, NNBOM's specific purpose is to enable the first large-scale, quantitative analysis of how NN software, as an ecosystem of these reusable components, evolves. Leveraging NNBOM, our study details the evolution of NN software through its scale, component reuse, and inter-domain dependencies.

III. Study Design

This section details the methodology employed to investigate the evaluation of NN software. Our primary objective is to construct a comprehensive NNBOM dataset, capturing the constituent components of NN repositories. Building upon this dataset, we conduct a large-scale empirical analysis to understand the evolutionary patterns of NN software. The resulting NNBOM dataset is made publicly available [15] to facilitate further research. The study design is structured as follows: First, we present the research questions (RQs) that guide our evolutionary analysis (Section III-A). Second, we describe the systematic process for constructing the NNBOM database, encompassing data collection from GitHub (Section III-B), the extraction and analysis of key NN components such as TPLs, pre-trained models (PTMs), and Modules (Section III-C), and subsequent data processing to prepare the NNBOM for our empirical study (Section III-D). Figure 2 provides an overview of this database construction pipeline.

A. Research Questions

To systematically analyze the evolution of NN software, we focus on three key facets: the changing scale of software, the reuse patterns of its core components, and the shifting landscape of inter-domain dependencies. These inquiries are formulated into the following research questions:

- RQ1 (Software Scale Evolution): How has the scale of NN software, particularly in Pytorch-based repositories, evolved in open-source communities, considering the number of versions, TPLs, PTMs, modules, and lines of code?
- RQ2 (Component Reuse Evolution): How have component co-usage patterns evolved in neural network software, and what structural trends can be observed in the long-term evolution of core Modules?
- RQ3 (Inter-Domain Dependency Evolution): How have inter-domain dependencies, evidenced by module reuse across different AI application domains, evolved within the NN software ecosystem?

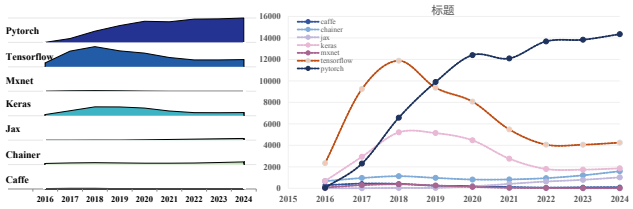

Fig. 3: Trends of deep learning framework usage over years: (Right) absolute repository counts; (Left) normalized proportions across frameworks.

The methodologies for addressing these RQs, along with their corresponding results and findings, are detailed in Sections IV, V, and VI, respectively.

B. Data Collection

Repository Collection Scope and Rationale. Following prior mining software repositories works [21]–[24], we collected open-source repositories from GitHub, the most popular platform for hosting open-source projects. A critical decision in defining our scope was the choice of the NN framework. The adoption of PyTorch has surged, with its usage in research papers growing from 29% in 2019 to 65% in 2023 [25], and it now underpins over 70% of AI research implementations [26]. As illustrated in Figure 3, which shows the trends of deep learning framework usage, the number of PyTorch-related repositories has consistently grown since 2018, while repositories for other frameworks have generally seen a decline or stagnation. Given PyTorch's clear dominance and to ensure a focused analysis by avoiding framework-specific inconsistencies that could introduce noise into evolutionary patterns, we concentrated our collection on PyTorch repositories. This approach allows for a deep dive into the most active and representative segment of the current NN open-source ecosystem. Our data collection process follows the method outlined in OSSFP [24]. Specifically, we use the official GitHub API to retrieve all repositories tagged with Python and containing the keyword "pytorch" in their topics or repository names. This initial query yielded 78,243 repository links with their associated metadata, such as topics, creation date, and fork count.

Filtering rules. To curate a dataset representative of substantive NN software evolution, we applied a multi-stage filtering process:

1) **Exclusion of Tutorial Projects:** Repositories with names or descriptions contraining keywords like "tutorial", "example", or "demo" were excluded. These often contain simplified, non-production code unsuited for studying real-world software evolution.
2) **Removal of Trivial Repositories:** Projects lacking any identifiable NN modular code components (as defined in Section III-C) were discarded, as they do not contribute to the component-level analysis of NNBOM.
3) **Quality and Activity Filtering:** We employed a standard repository ranking tool [27], to assess project activity and maintenance. The bottom 5% of repositories based on this score[1] were filtered out to minimize the impact of inactive or poorly maintained projects.

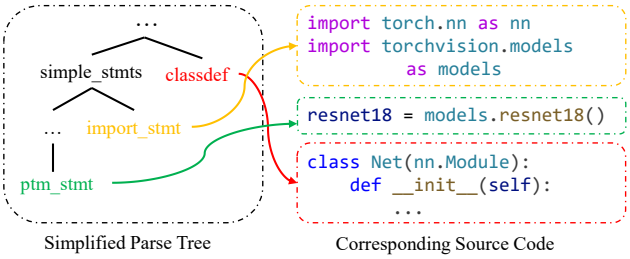

Fig. 4: Extract specific components from the parse tree.

After applying these filters, our dataset comprised 55,997 PyTorch repositories, forming the basis for NNBOM construction.

Version collection. For each selected repository, we collected version information. Since most GitHub repositories mark releases or significant milestones using Git tags, we extracted these tags as distinct versions. For repositories without any Git tags, we designate their current main branch as a single representative version. All versions were downloaded via the GitHub API, resulting in a total of 93,647 distinct repository versions for analysis. Data collection scripts and additional repository details are available at [15].

C. Component Analysis

The NNBOM for each repository is constructed by identifying and extracting three primary categories of NN software components: TPLs, PTMs, and neural network modules. Figure 4 illustrates how these components can be identified from source code, often via specific nodes in a parsed Abstract Syntax Tree (AST). The subsequent subsections detail the extraction process for each component type.

1) TPL Extraction.

TPLs are pre-packaged Python libraries, typically distributed via PyPI or WPILib, that provide essential functionalities. We extract TPL information using the two approaches below:

- **Extraction from Configuration Files:** We parse project configuration files (e.g., "setup.py", "requirements.txt") which often declare TPL dependencies and their versions. However, as noted by Peng *et al.* [28], these files can be incomplete.
- **Scanning Import Statements:** We use ANTLR to parse Python source code and identify all `import` statements. This captures TPLs used directly in code, though `import` statements typically do not specify version information.

To mitigate the limitations of each method, we combine these methods. Information from configuration files (including versions) is prioritized. The `import` statements are analyzed to distinguish between local project modules (e.g., relative imports like `import .module` or import matching project subdirectories) and external TPLs. The identified external TPLs from `import` statements are then merged with those from

[1]We chose 5% as a practical trade-off, which should be sufficient to eliminate low-quality repositories without significantly affecting the validity of large-scale evolution analysis

configuration files to create a comprehensive list for each repository version. This combined approach provides a more accurate depiction of TPL usage.

2) PTM Extraction.

To save computational resources and leverage existing models, developers often reuse PTMs. We detect PTM invocations, particularly those from model hubs, by:

- **Identifying Model Hub APIs:** Following Jiang *et al.* [29], we target PTM invocations from eight popular model hubs: Hugging Face [30], TensorFlow Hub [31], Model Zoo [32], PyTorch Hub [33], ONNX Model Zoo [34], Modelhub [35], NVIDIA NGC [36], MATLAB Model Hub [37]. Six of these hubs provide APIs for PTM invocation; the other two (ModelZoo, ONNX Model Zoo) involve repository cloning, detected by comparing them against a database.
- **Recognizing LLM Deployment Frameworks:** We also identify PTM usage via large language model (LLM) deployment frameworks like vLLM [38], DeepSpeed-MII [39], and CTranslate2 [40]. An example of such an invocation is shown in Figure 1 (item ④).
- **Customized Parsing:** We modified Python's lexical ("lexer.g4") and grammar ("parser.g4") files to define rules specific to each hub's and framework's PTM invocation patterns. These customized grammars are used to build ASTs. During tree traversal, a symbol table tracks variable assignments. When a PTM invocation is identified, its path parameter is resolved (either directly if a string or via the symbol table if a variable) to capture the PTM source.

This systematic approach yields a detailed list of PTM invocations within the analyzed projects.

3) Module Extraction

In the PyTorch framework, an NN module is any class that inherits, directly or indirectly, from "torch.nn.Module". These are the fundamental building blocks of NN architectures. We extract these modules as follows:

When starting with a project, the initial step involves creating a symbol table and adding "torch.nn.Module" as an entry. Following this, we generate a parser tree for each Python file to aid in identifying classes and mapping their inheritance relationships. As we navigate through the classes in the project, any class whose superclass is already listed in the symbol table is also added to the table. This process is repeated until there are no further changes to the size of the symbol table, indicating that all relevant classes have been captured. The classes captured in the symbol table, other than "torch.nn.Module", are recognized as the NN modules of the project. To improve parsing efficiency, particularly when dealing with multiple versions of the same repository that contain numerous identical files, we implement an incremental parsing approach. Only files that differ between versions are parsed for each new version. The differences between consecutive versions are determined using the "git log" command, which allows for faster processing while maintaining accuracy.

4) Effectiveness of Component Extraction

To evaluate the effectiveness of our component extraction pipeline, we conducted a manual validation study on a random sample of 100 repositories, which aligns with prior work [41]–[43]. Typically, about 100 repositories are randomly selected for manual inspection to balance representativeness and manageable workload. Specifically, we randomly selected 100 repositories stratified into five bins based on the total number of extracted modules, with 20 repositories sampled from each bin. This stratified sampling ensures coverage across repositories of varying complexity. By comparing the manually annotated results with those generated by our method, we found that the identification accuracy for third-party libraries (TPLs) and neural network modules both reached 100%, correctly identifying 2,431 TPLs and 1,271 neural network modules. The identification accuracy for pretrained models (PTMs) was slightly lower at approximately 98%, with only 2 out of 83 instances missed due to dynamic specification of model information via console input, which is inherently unobservable through static analysis. Overall, the extraction method achieved a success rate of approximately 99.9%, providing a strong foundation for the subsequent evolutionary analysis. Since the primary objective of this study is to empirically investigate the neural network software ecosystem, detailed evaluation results are omitted. The complete source code and the manually verified dataset are available at [15].

TABLE I: Normalization rules

Rule	Description
remove	Remove all comments and whitespace, including blank lines
rename	Standardize variable names to a consistent naming convention
replace	Replace numeric and string literals with specified placeholders

D. Data Processing

Once the raw component information (TPLs, PTMs, Modules) is extracted for all repository versions, further processing is necessary to construct the final NNBOM database and prepare it for evolutionary analysis. This involves module normalization and clone detection for dependency construction, index building for efficient data retrieval, and feature generation for both versions and modules.

1) Module-based Dependency Construction

To understand component reuse and infer inter-repository dependencies, we identify cloned NN modules. A module in repository A is considered a reuse of a module from repository B if it is detected as a clone and appears later in time, following the timestamp-based inference standard in prior studies [44], [45]. Our study focuses on detecting Type-1 (exact textual clones) and Type-2 (syntactic clones), as these forms of reuse are both prevalent and reliable. Empirical evidence confirms that Type-1/2 clones constitute the dominant types of reuse in open-source software [46], while Type-3 and Type-4 clones have less consistent definitions and less reliable detection, posing a significant risk of introducing noise and false positives in large-scale analysis [47], [48]. By limiting our analysis to high-confidence Type-1/2 clones, we ensure that the findings on module reuse are both scalable and reliable. For clone detection, we adopt well-established tools [49]–[52], applying NN module-

specific normalization. These methods have demonstrated high accuracy in detecting Type-1/2 clones, thereby supporting the robustness of our database construction and subsequent empirical analysis. The normalization rules, detailed in Table I, are applied to each extracted NN module before hashing. These steps reduce superficial differences, allowing for more accurate identification of functionally similar or identical modules. After normalization, a hash value is generated for each module. Modules sharing the same hash are grouped into a "clone family". If two different repositories contain modules belonging to the same clone family, we consider a dependency link to exist between them, mediated by the shared module. Note that this mechanism establishes dependencies between repositories, not between different versions of the same repository.

2) Index Building

In this phase, we aim to establish indices for both modules and versions. By creating these indices, we can quickly retrieve various types of information about the version corresponding to each module. This enables us to efficiently analyze clone families, determining their application domains (by aggregating the domains of all modules), their lifespan (the difference between the earliest and latest module creation years), and more. Additionally, these indices allow us to swiftly obtain clone information between versions, facilitating the construction of the dependency graph.

3) Feature Generation

Finally, we generate a structured set of features for each version and each unique NN module in the NNBOM.

The version features are as follows:

- **Version Index.** The index of the version.
- **TPL List.** The TPLs that the version depends on. The TPLs extracted from configuration files contain version information, whereas those extracted from `import` statements do not.
- **PTM List.** The PTMs invoked by the version.
- **Self-Developed Module List.** The index of the modules developed independently within the version.
- **Cloned Module List.** The index of the cloned modules within the version.
- **Release Time.** The release time of the version. This feature is vital for determining the origin of the module and conducting an evolutionary analysis.

The module features are as follows:

- **Module Index.** The index of the module.
- **Module Hash.** The hash value of the normalized module. It ensures accurate mapping of module pairs without structural and semantic differences.
- **Version Index.** The version from which the module originates.
- **Lines of Code (LoC).** The number of code lines in the module, including blank lines and comments.
- **Domains.** To analyze the application scope of the AI software module, repositories are classified into seven representative domains: Unsupervised Learning (UL), Reinforcement Learning (RL), Computer Vision (CV), Multimodal Learning (MML), Natural Language Processing (NLP), Generative Models (GM), and Transformer (Trans). These domains were selected by jointly considering top categories from the Model Zoo platform [32] and prevalent AI research areas. For accurate classification, a list of representative keywords for each domain was manually curated with domain expert assistance (detailed in our replication package [15]). A repository is assigned to a domain if its name or GitHub topics match any of these keywords. Repositories can be assigned to multiple domains if they match keywords from several categories, or to no domain if relevant metadata is insufficient or missing. Modules inherit the domain(s) of their parent repository.
- **Module Frequency.** The number of times the module has been cloned in the GitHub community. A higher number of clones indicates a greater popularity of the module.

TABLE II: Annual changes in overall trends

Year	Total				Average	
	Version	TPL	PTM	Module	Module	LoC
2017	1,414	49,045	131	20,454	14.47	54.09
2018	5,089	138,535	981	42,059	8.26	49.95
2019	8,737	222,860	1,031	107,275	12.28	49.77
2020	12,833	291,988	2,154	202,197	15.76	50.57
2021	15,267	299,193	2,393	440,779	28.87	52.35
2022	16,468	317,097	3,641	512,857	31.14	55.75
2023	16,684	290,761	5,475	685,572	37.29	56.57
2024	17,155	285,284	7,753	925,296	62.58	46.81

*4) Overview of the Constructed **NNBOM Dataset***

Upon completion of the data collection, component analysis, and processing pipeline, the final NNBOM dataset is compiled. This dataset encompasses detailed component information for each repository version and an inter-repository dependency graph. In this graph, nodes represent individual repositories, and weighted edges signify the number of shared module clone families between them, derived as described in Section III-D1. The resulting NNBOM dataset provides a comprehensive view of component-level composition and inter-repository dependencies within the studied PyTorch ecosystem. It comprises data from 93,647 repository versions originating from 55,997 curated repositories. This dataset includes a total of 1,894,763 TPLs, 23,559 PTM invocations, and 3,102,311 NN modules. Across these repositories, we identified 2,135,351 version dependency edges based on the shared module clones, highlighting extensive component reuse and cross-project propagation. This rich NNBOM dataset serves as the foundation for the empirical evolution analysis presented in the subsequent sections.

IV. RQ1: Software Scale Evolution

A. Analysis Method

To address **RQ1**, we first analyze the overall evolutionary trend of neural network software across dimensions: the total number of repository versions (*Total Version*), the count of distinct third-party libraries (*Total TPL*), the number of invoked pretrained models (*Total PTM*), and the total count of modules (*Total Module*). Subsequently, we examine the scale characteristics of modules, the core NNBOM component, by analyzing the average number of modules per version (*Average Module*) and the average lines of code per module (*Average LoC*). Additionally, we investigate the distribution of version

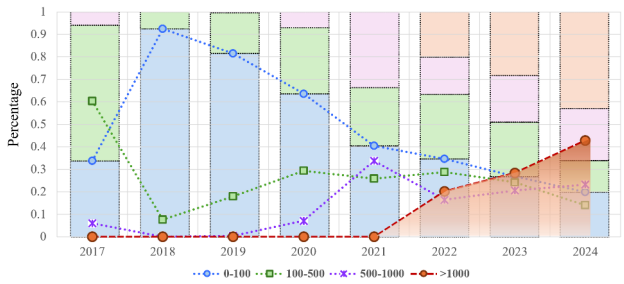

Fig. 5: Proportional changes in versions of different sizes.

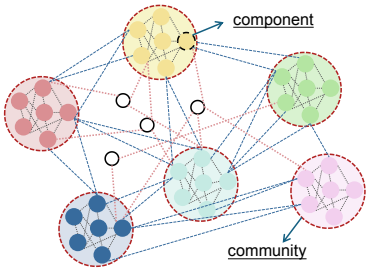

Fig. 6: Example of a component co-occurrence network.

sizes, based on module counts, to capture structural variation across repositories.

B. Results and Findings

Overall Trend. Table II (Total column) presents the overall development trend within the PyTorch community. The first column reveals a consistent increase in neural network versions since PyTorch's 2016 release, highlighted by a remarkable growth of 807.57% from 2017 to 2020, signifying a period of rapid expansion. The subsequent columns in Table II detail the evolution of each NNBOM component. Specifically, the number of distinct TPLs utilized per year increased substantially before 2020. After this period of rapid expansion, the annual growth in the number of TPLs used across the ecosystem significantly slowed, with overall usage patterns stabilizing, indicating a maturation of the TPL ecosystem. In contrast, the total number of PTM invocations and NN modules have both continued to grow steadily. Although overall PTM usage has increased, the number of direct PTM invocations remains relatively low; further analysis reveals that only 7.6% of repositories include such invocations. Considering the extensive presence of clone modules identified during our dataset construction, it is evident that *researchers often prefer cloning models over directly invoking PTMs when developing neural network systems*. This preference likely reflects a desire for greater flexibility and control in model customization.

> **Finding 1:** All NNBOM components exhibit continuous growth in scale. TPLs experienced rapid expansion until 2020, after which their usage stabilized, indicating a maturation of the TPL ecosystem. Meanwhile, PTMs and Modules show sustained growth.

Intra-Project Module Scale and Size. Since a module is essentially a self-contained code block, its total count is naturally influenced by the increasing number of repository versions. To decouple this effect and understand project-level characteristics, we compute the average number of modules per version and the average lines of code (LoC) per module, as shown in Table II (Average columns). The former metric reflects the module scale of each version, while the latter captures the average size of a module. From 2018 onward (excluding 2017 due to early-stage data instability), the average number of modules per version has steadily increased, indicating a clear trend toward larger project sizes. Conversely, the average LoC per module has remained relatively stable throughout the study period, consistently hovering around 50 lines. This stability suggests that as neural network tasks grow in complexity, developers tend to decompose functionality into a greater number of smaller modules, rather than constructing larger, monolithic modules. The consistent module size of around 50 lines may reflect an implicit developer preference or convention for module granularity. Figure 5 provides a more detailed view, illustrating the annual distribution of repository versions categorized by their module count. Here, we categorize version sizes into four intervals: *0–100*, *100–500*, *500–1000*, and >1000 modules. Except for the early-stage year 2017, a clear decline is observed in smaller versions (0–100) starting from 2018, suggesting the gradual maturation of projects beyond initial prototypes. Versions containing 100–500 and 500–1000 modules have become increasingly prevalent over time. Notably, from 2022 onward, there has been a sharp rise in the proportion of large-scale versions (>1000 modules). This surge coincides with the emergence of large language models and marks the onset of a new era in ultra-scale neural network development. This trend highlights *a substantial shift toward building larger and more architecturally complex models*, driven by field advances and increased computational capabilities.

> **Finding 2:** The increasing average number of modules per version, coupled with a stable average module LoC (around 50 lines), indicates a consistent development convention: developers are addressing growing task complexity by decoupling projects into a greater number of smaller modules.

V. RQ2: Component Reuse Evolution

A. Analysis Method

To address RQ2, we design a two-fold exploratory analysis to examine the reuse evolution of neural network components. First, as illustrated in Figure 6, we construct annual component co-occurrence networks. Each node in these networks represents a unique NNBOM component, and edges indicate strong co-usage, defined as co-occurrence in at least five different repositories. We then apply the Louvain algorithm [53] to partition these networks, maximizing modularity to identify

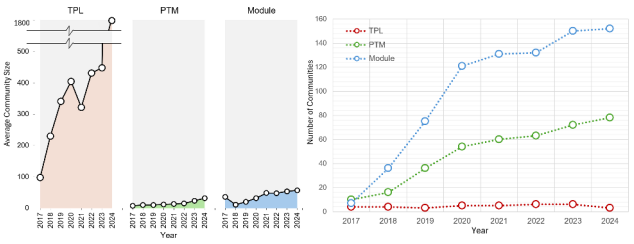

Fig. 7: Yearly trends of co-usage community count and average size, reflecting the evolving structure of component reuse.

communities. Each community represents a cluster of components frequently used in conjunction. By tracking the number and average size of these communities over time, we analyze the evolution of co-usage patterns and the structural organization of component dependencies. Second, we focus on Modules, the core components of neural networks, and examine how the most frequently reused ones evolve over time. This analysis aims to uncover shifts in architectural preferences and highlight emerging functional trends in neural network design.

B. Results and Findings

Co-usage Community Dynamics. Figure 7 illustrates the temporal trends in both the number and average size of co-usage communities for each component type. First, the average community size exhibits a consistent upward trend across all component types, with TPLs exhibiting the fastest growth. This indicates increasing complexity in the co-occurrence networks, where components become more interconnected and form larger co-usage clusters. The pattern reflects a growing tendency for components, especially TPLs, to be reused alongside a broader set of counterparts, driven by accumulated developer experience and the gradual convergence of reusable modules into larger, and more integrated communities. Second, the number of communities displays divergent trends. Both PTMs and Modules show sustained growth in community count, with Modules increasing most rapidly. This suggests that, alongside forming larger clusters, these components continue to diversify by supporting emerging functionalities and spawning new co-usage groups. In contrast, the number of TPL communities remains relatively stable over time. This stability, combined with growing community size, implies that TPLs are maturing into stable, general-purpose clusters with fewer novel configurations emerging.

> **Finding 3:** All NNBOM components are increasingly integrated into larger co-usage communities, while PTMs and Modules additionally drive the emergence of new functional clusters.

Functional Evolution of Modules. To examine the functional evolution of neural network Modules, we construct a temporal evolution tree based on the top ten most reused Modules for each year from 2017 to 2024. These Modules reflect the dominant architectural patterns of each era. Their transitions reveal three key evolutionary phases, as shown in Figure 8.

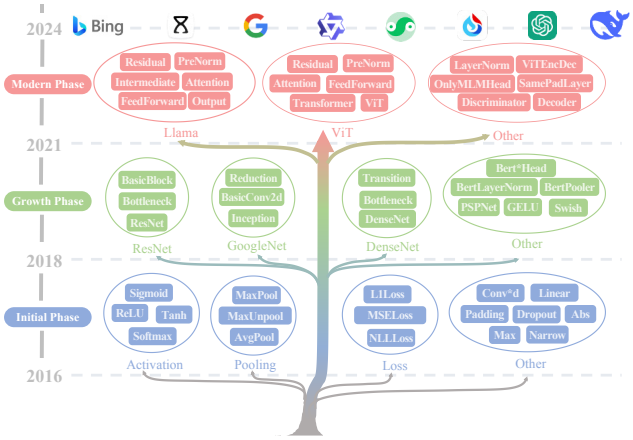

Fig. 8: Evolution Tree: Tracing the Development of Neural Network.

Initial Phase (2017). This stage features frequent reuse of simple functional Modules such as `sigmoid` and `padding`, reflecting PyTorch's early requirement to encapsulate even basic operations within `nn.Module`.

Growth Phase (2018–2020). CNN-based architectures, especially ResNet and Inception, become mainstream. Their components are widely reused, often through direct cloning rather than invoking pre-trained models, enabling structural customization.

Transformer Expansion Phase (2021–2024). This phase witnesses a surge in Transformer-based Modules and increasing structural diversity. Developers increasingly build upon existing Transformer infrastructures, exemplified by the rise of modules such as `OnlyMLMHead` and `Attention` blocks. These Transformer-centric modules mark the beginning of the large model era, underpinning the development of a new wave of commercial-scale pre-trained systems.

> **Finding 4:** Module reuse has evolved from simple functional operations to Transformer-centric architectures, reflecting a structural transition toward scalable designs that underpin the emergence of modern large-model systems.

VI. RQ3: Inter-Domain Dependency Evolution

A. Analysis Method

To address **RQ3**, we conduct a three-part analysis focusing on: (1) annual changes in average entropy, (2) detailed trends in inter-domain dependency, and (3) the correlation between module lifespan and its domain usage range.

First, we calculate the annual average entropy to uncover the overall trend in inter-domain module dependency over time.

TABLE III: Annual changes in average entropy

Year	2017	2018	2019	2020	2021	2022	2023	2024
Average Entropy	0.157	0.175	0.231	0.265	0.298	0.385	0.457	0.485

The average entropy $\bar{H}$ for each year is computed as follows:

$$\bar{H} = \frac{1}{N}\sum_{i=1}^{N} H_i, \quad H_i = -\sum_{k=1}^{K} p_k \log(p_k) \tag{1}$$

where N is the number of clone families in a given year, H_i denotes the entropy of the i-th clone family, and p_k is the proportion of modules in the k-th domain. A higher $\bar{H}$ indicates greater overall diversity in how module families span across different domains, suggesting increased inter-domain reuse.

After identifying the overall trend, we analyze specific inter-domain dependencies by examining module overlap among our seven representative domains. We identify the top five domain pairs with the highest degree of module sharing each year. The degree of overlap is quantified using:

$$P_{AB} = \frac{|A \cap B|}{|A \cup B|} \tag{2}$$

where A and B represent two domains, and P_{AB} indicates the proportion of shared modules between them.

Finally, we investigate the correlation between a module's lifespan and its domain usage range. Here, a module's lifespan is defined as the number of years between the earliest and latest versions in its clone family, while the domain range is measured by the number of distinct domains in which the family appears.

B. Results and Findings

Annual Changes in Average Entropy. Table III presents the annual average entropy of domain diversity for module clone families. A higher entropy value signifies greater inter-domain module reuse. The data reveals a consistent upward trend in average entropy from 0.157 in 2017 to 0.485 in 2024, indicating increasingly frequent and diverse reuse of module clone families across different AI domains. This trend suggests an evolving design philosophy: modules embodying general-purpose, suitable for reuse across diverse domains, are becoming more prevalent compared to highly task-specific implementations. This shift points towards a growing emphasis on modularity and reusability driven by the inherent agnosticism of foundation NN components to specific tasks.

Finding 5: Rising entropy suggests increasing cross-domain reuse and a shift toward general-purpose modules.

Details of Inter-Domain Dependency. The shared module proportions across domain pairs offer empirical insight into the architectural evolution of neural network software. Table IV presents the top five domain pairs with the highest frequency of cross-referenced modules each year. In 2017, the top-ranked pairs were UL-GM (17.54%), CV-GM (12.14%), and CV-UL (7.33%), indicating that module reuse was initially concentrated in unsupervised learning, generative modeling, and computer vision. The overlap among these three domains reflects a characteristic of the **early stage**: the dominance of GAN-based unsupervised image generation frameworks, primarily applied to visual tasks. Notably, NLP-related structures were absent from the top reuse pairs, as large-scale text generation had not yet emerged as a major trend.

TABLE IV: Top 5 domain pairs by shared modules

Year	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5
2017	U-G-17.54	C-G-12.14	C-U-7.33	N-G-5.98	C-N-4.68
2018	C-G-14.02	T-N-12.52	G-M-5.03	C-N-3.95	N-G-2.84
2019	T-N-16.25	U-G-11.07	C-G-10.63	N-R-6.37	T-C-6.09
2020	T-N-21.31	T-C-8.23	C-G-8.17	U-G-6.89	C-N-6.23
2021	T-N-25.34	T-C-12.13	C-N-6.45	C-G-5.37	T-M-3.94
2022	T-N-27.87	T-C-19.36	C-N-8.79	T-M-7.88	N-M-4.61
2023	T-N-29.86	T-C-19.94	T-G-12.23	T-M-10.08	C-N-9.01
2024	T-N-29.56	N-G-21.16	T-G-17.25	T-M-16.32	T-C-10.68

Note. $\mathcal{D}_1$–$\mathcal{D}_2$–$\rho_{\mathcal{D}_1\mathcal{D}_2}$ indicates that $\rho_{\mathcal{D}_1\mathcal{D}_2}\%$ of modules are shared between domains $\mathcal{D}_1$ and $\mathcal{D}_2$.

From 2018 onward, Transformer modules (Trans) emerged and rapidly gained prominence. The T-N (Transformer + NLP) pair immediately entered the top five in 2018 with a 12.52% share and ranked first every year thereafter, peaking at 29.86% in 2023. This sharp rise highlights the central role of NLP in reusing Transformer-based architectures, in line with the emergence of attention-based language models. This marks the beginning of the **Transformer stage** in 2018. Following this, the Transformer architectural paradigm began to diffuse into other domains. The diffusion trajectory unfolded in distinct phases: The first major wave occurred in computer vision. The T-C (Transformer + CV) pair appeared in 2020 (8.23%) and steadily increased to 19.94% by 2023, driven by the adoption of Vision Transformers (ViT) and Transformer-based object detection models such as DETR. This trend reflects a structural migration in CV toward Transformer-based backbones. A second wave of diffusion emerged in the domains of generative modeling and multimodal learning. The T-M (Transformer + Multimodal) pair rose from 7.88% in 2022 to 16.32% in 2024, while T-G (Transformer + GM) entered the top five in 2023 and reached 17.25% in 2024. These shifts correspond to the rise of large-scale language models and a transition in generative modeling from image-centric synthesis to Transformer-based generation. Most notably, by 2024, the N-G (NLP + GM) pair surpassed the traditional C-G (CV + GM), which dropped out of the top five. This indicates a new convergence: generative tasks are increasingly grounded in NLP-centric, Transformer-driven architectures, reflecting a realignment of generative AI around language-first paradigms. Together, these trends depict a multi-phase diffusion trajectory of the Transformer paradigm: from its NLP origins to widespread structural integration across vision and generative tasks, *culminating in the convergence of language and generation as the dominant axis in modern neural network development.*

TABLE V: Module count by lifespan and domain

L.\ D.	1	2	3	4	5	6	7
1	74373	21030	3553	673	54	2	0
2	7515	5214	1833	343	41	3	0
3	1863	2530	869	526	289	30	4
4	826	1016	483	399	52	14	2
5	397	547	254	189	84	25	0
6	134	157	172	51	40	18	5
7	17	53	66	27	17	40	8
8	0	3	16	4	6	4	2

L.: Lifespan (number of years); D.: Domain Range.

Finding 6: Cross-domain dependencies evolved from early GAN-based visual generation to a Transformer-driven phase marked by widespread architectural diffusion and a shift toward text generation.

Correlation Between Module Lifespan and Domain Range. The third perspective shifts focus to the relationship between module lifespan and cross-domain usage. Table V presents the number of modules categorized by their lifespan (i.e., number of years observed) and domain range (i.e., number of distinct domains in which they are used). The results reveal a clear positive correlation between lifespan and domain range: modules reused across more domains tend to have significantly longer lifespans. For example, when the domain range is limited to one, 74,373 modules have a lifespan of only one year, while none survive for eight years. This pattern demonstrates that most modules are both short-lived and domain-specific, with counts decreasing sharply as lifespan increases. Conversely, fixing the lifespan to one year (i.e., the first row), the number of modules drops rapidly as domain range increases, further confirming that broadly reused modules are rarely short-lived. This bidirectional trend suggests that the ability to generalize across domains is a strong indicator of a module's utility and longevity. To illustrate this pattern, consider the most notable modules that exhibit both the longest lifespan of eight years and usage across up to seven domains. Although rare, these modules are highly representative. According to NNBOM, they include `Bottleneck` and `BasicBlock`, two foundational building blocks introduced in ResNet and widely reused as backbones across a variety of neural architectures. Their sustained presence across time and domains reflects both their functional robustness and broad architectural compatibility.

Finding 7: A wider domain range indicates stronger functionality and broader applicability, often correlating with longer lifespans. Cross-domain adaptability thus serves as a promising and quantifiable indicator of module quality and long-term stability.

VII. Application

To demonstrate the practical utility of the NNBOM dataset and our analytical findings, we developed two prototype applications. These prototypes showcase how NNBOM can directly support software engineering tasks for both ecosystem maintainers and individual developers.

(1) Multi-repository Evolution Analyzer. This tool enables maintainers to dynamically assess the impact of newly added repositories on the broader neural network software ecosystem. It quantifies growth in TPLs, PTMs, and modules, as well as the expansion of module-level dependencies by distinguishing between reused and newly created modules. This offers a practical lens into the ongoing evolutionary dynamics of the ecosystem.
(2) Single-repository Component Assessor and Recommender. Aimed at developers, this tool analyzes a target repository using NNBOM to assess its components, highlighting outdated or newly added modules. It also recommends potentially useful components based on co-usage patterns and identifies similar repositories based on overall component similarity.

Case Study. (1) For multi-repository analysis, we examine repositories added in December 2024 using the NNBOM constructed from data before November 2024. The tool identifies 94 previously unseen TPLs, 44 new PTMs, and 6,545 new Modules introduced by these repositories. Among a total of 14,467 Modules in this batch, 45.2% are original creations, highlighting a notable level of innovation.
(2) For single-repository analysis, based on the API search for top-recommended repositories created after January 1, 2025, the repository `test-time-training/ttt-video-dit` stands out. Created on April 13, 2025, it has quickly gained 1.6k stars and 133 forks. This repository contains 43 TPLs, no PTMs, and 40 modules, of which 32 are newly created and eight are reused. Among the reused modules, the oldest dates back to 2020, while 4 were introduced in 2024, indicating that reused modules typically originate from the preceding year. Notably, it introduces a new TPL, `test-time-training`, released on February 11, 2025, which provides support for test-time training kernels. According to our similarity analysis based on module composition, the most similar repository is `Auto1111SDK/Auto1111SDK`, suggesting potential architectural or functional overlap that developers can reference or further explore.

Detailed usage instructions and case studies are provided in [15].

VIII. Threats to Validity

A. Internal Validity

Domain Labeling Accuracy. To assign domain labels for RQ3, we relied on keywords in repository names and topic tags. This heuristic, while capturing many cases, may overlook repositories with ambiguous or missing metadata. potentially introducing error. However, fully reliable automated domain inference from content is challenging and remains unavailable. Our approach, prioritizing precision via manually curated keywords (detailed in Section III-D3 and [15]), minimizes noise in the inter-domain analysis.

Toolchain Reliability. Another potential threat concerns the effectiveness of the toolchain used during dataset construction. Metadata collection is straightforward. Module dependencies for RQ3 rely on Type-1/2 clone detection, previously validated for high accuracy [49]–[52] and applied with carefully designed

normalization (Table I) to ensure precision. The primary potential bias is in component extraction (TPLs, TPMs, Modules). Our manual validation (Section III-C4) on a stratified sample of 100 repositories showed high overall accuracy (∼99.9%), with minor PTM identification errors (2 out of 83 due to dynamic runtime specification, a static analysis limitation) having negligible impact on dataset quality. For untagged repositories, using the main branch as a single version is a simplification that might underrepresent their evolution for RQ1, but ensures their inclusion; the majority of our versions derive from tags, mitigating this.

B. External Validity

Our analysis focuses exclusively on PyTorch repositories, which may not fully represent the entire NN software ecosystem. Other frameworks were excluded due to significant methodological challenges in cross-framework module normalization and PTM identification. Components often lack direct, reliable counterparts across frameworks, making unified analysis beyond our current scope. However, as discussed in Section III-B, PyTorch is the dominant framework in AI research and open-source development. Thus, our focus on 55,997 curated PyTorch repositories, selected via rigorous filtering to include substantive projects, provides a strong and representative basis for analyzing contemporary NN software evolution, minimizing generalization bias within this leading ecosystem.

IX. Conclusion

In this study, we construct NNBOM, a large-scale dataset capturing the component composition and dependencies of 55,997 neural network repositories. Based on this foundation, we conduct a comprehensive empirical analysis of neural network software evolution across three dimensions: software scale, component reuse, and inter-domain dependencies. Our findings offer valuable insights into the structural and functional trends shaping open-source neural network development.

Acknowledgments

This research is supported in part by the National Defense Basic Scientific Research Program of China (grant No. JCKY2023210C009) and the National Natural Science Foundation of China (grant No. 61972373).

References

[1] S. Carmody, A. Coravos, G. Fahs, A. Hatch, J. Medina, B. Woods, and J. Corman, "Building resilient medical technology supply chains with a software bill of materials," *npj Digital Medicine*, vol. 4, no. 1, pp. 1–6, 2021.

[2] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt *et al.*, "Towards fully autonomous driving: Systems and algorithms," in *2011 IEEE intelligent vehicles symposium (IV)*. IEEE, 2011, pp. 163–168.

[3] M. Kim and D. Notkin, "Using a clone genealogy extractor for understanding and supporting evolution of code clones," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[4] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 485–495.

[5] L. Barbour, L. An, F. Khomh, Y. Zou, and S. Wang, "An investigation of the fault-proneness of clone evolutionary patterns," *Software Quality Journal*, vol. 26, pp. 1187–1222, 2018.

[6] P. Thongtanunam, W. Shang, and A. E. Hassan, "Will this clone be short-lived? towards a better understanding of the characteristics of short-lived clones," *Empirical Software Engineering*, vol. 24, pp. 937–972, 2019.

[7] M. Assi, S. Hassan, and Y. Zou, "Unraveling code clone dynamics in deep learning frameworks," *arXiv preprint arXiv:2404.17046*, 2024.

[8] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 102–112.

[9] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 351–361.

[10] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[11] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, S. Zhong, B. Yin, and X. Hu, "Harnessing the power of llms in practice: A survey on chatgpt and beyond," *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 6, pp. 1–32, 2024.

[12] B. K. Chan, "Artificial intelligence bill of materials (aibom)," https://minddata.org/bill-of-artificial-intelligence-materials-boaim-Brian-Ka-Chan-AI, 2024.

[13] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu, "An empirical study on software bill of materials: Where we stand and the road ahead," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2630–2642.

[14] T. Stalnaker, N. Wintersgill, O. Chaparro, M. Di Penta, D. M. German, and D. Poshyvanyk, "Boms away! inside the minds of stakeholders: A comprehensive study of bills of materials for software systems," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[15] NNBOM, "Nnbom," https://github.com/NNBOM24/nnbom24, 2024.

[16] S. Carmody, A. Coravos, G. Fahs, A. Hatch, J. Medina, B. Woods, and J. Corman, "Building resilient medical technology supply chains with a software bill of materials," *npj Digital Medicine*, vol. 4, no. 1, pp. 1–6, 2021.

[17] B. Xia, D. Zhang, Y. Liu, Q. Lu, Z. Xing, and L. Zhu, "Trust in software supply chains: Blockchain-enabled sbom and the aibom future," *arXiv preprint arXiv:2307.02088*, 2023.

[18] Wiz.io, "AI BOM: AI Bill of Materials," https://www.wiz.io/academy/ai-bom-ai-bill-of-materials, 2024, accessed: 2025-05-19.

[19] Manifest Cyber, "AIBOM – The AI Bill of Materials," https://www.manifestcyber.com/aibom, 2024, accessed: 2025-05-19.

[20] K. Bennet, G. K. Rajbahadur, A. Suriyawongkul, and K. Stewart, "Implementing ai bill of materials (ai bom) with spdx 3.0: A comprehensive guide to creating ai and dataset bill of materials," *arXiv preprint arXiv:2504.16743*, 2025.

[21] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 155–165.

[22] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *2016 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2016, pp. 334–344.

[23] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, pp. 3219–3253, 2017.

[24] J. Wu, Z. Xu, W. Tang, L. Zhang, Y. Wu, C. Liu, K. Sun, L. Zhao, and Y. Liu, "Ossfp: Precise and scalable c/c++ third-party library detection using fingerprinting functions," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 270–282.

[25] Y. Hao, X. Zhao, B. Bao, D. Berard, W. Constable, A. Aziz, and X. Liu, "Torchbench: Benchmarking pytorch with high api surface coverage," *arXiv preprint arXiv:2304.14226*, 2023.

[26] PyTorch Team, "2024 year in review," https://pytorch.org/blog/2024-year-in-review/, 2024, accessed: 2025-03-30.

[27] O. S. S. Foundation, "Criticality score," 2023, accessed: 2023-12-10. [Online]. Available: https://github.com/ossf/criticality_score/tree/main

[28] Y. Peng, R. Hu, R. Wang, C. Gao, S. Li, and M. R. Lyu, "Less is more? an empirical study on configuration issues in python pypi ecosystem," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[29] W. Jiang, N. Synovic, R. Sethi, A. Indarapu, M. Hyatt, T. R. Schorlemmer, G. K. Thiruvathukal, and J. C. Davis, "An empirical study of artifacts and security risks in the pre-trained model supply chain," in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, 2022, pp. 105–114.

[30] H. Face, "Hugging face – the ai community building the future." https://huggingface.co/, 2021.

[31] TensorFlow, "Tensorflow hub," https://www.tensorflow.org/hub, 2024.

[32] Y. K. Jing, "Model zoo - deep learning code and pretrained models," https://modelzoo.co/, 2024.

[33] PyTorch, "Pytorch hub," https://pytorch.org/hub/, 2024.

[34] ONNX, "Onnx model zoo," https://github.com/onnx/models, 2024.

[35] C. Imaging and B. Lab, "Modelhub," http://modelhub.ai/, 2024.

[36] NVIDIA, "Nvidia ngc: Ai development catalog," https://catalog.ngc.nvidia.com/, 2024.

[37] MathWorks, "Matlab deep learning model hub," https://www.mathworks.com/solutions/deep-learning.html, 2024.

[38] vLLM, "vllm," https://docs.vllm.ai/en/latest/, 2024.

[39] Microsoft, "Deepspeed model implementations for inference (mii)," https://www.microsoft.com/en-us/research/project/deepspeed/deepspeed-mii/, 2024.

[40] CTranslate2, "Ctranslate2," https://opennmt.net/CTranslate2/index.html, 2024.

[41] N. Lambaria and T. Cerny, "A data analysis study of code smells within java repositories," *Annals of Computer Science and Information Systems*, vol. 32, 2022.

[42] S. Idowu, Y. Sens, T. Berger, J. Krüger, and M. Vierhauser, "A large-scale study of ml-related python projects," in *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, 2024, pp. 1272–1281.

[43] E. Sülün, "An empirical analysis of issue templates on github," Ph.D. dissertation, PhD thesis. bilkent university, 2023.

[44] G. Zhang, X. Peng, Z. Xing, S. Jiang, H. Wang, and W. Zhao, "Towards contextual and on-demand code clone management by continuous monitoring," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 497–507.

[45] M. Jahanshahi, D. Reid, and A. Mockus, "Beyond dependencies: The role of copy-based reuse in open source software development," *ACM Transactions on Software Engineering and Methodology*, 2024.

[46] S. Feng, W. Suo, Y. Wu, D. Zou, Y. Liu, and H. Jin, "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*, 2024, pp. 1–13.

[47] C. K. Roy and J. R. Cordy, "Benchmarks for software clone detection: A ten-year retrospective," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 26–37.

[48] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 70–80.

[49] S. Feng, W. Suo, Y. Wu, D. Zou, Y. Liu, and H. Jin, "Machine learning is all you need: A simple token-based approach for effective code clone detection," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 222:1–222:13. [Online]. Available: https://doi.org/10.1145/3597503.3639114

[50] Y. Wang, Y. Ye, Y. Wu, W. Zhang, Y. Xue, and Y. Liu, "Comparison and evaluation of clone detection techniques with different code representations," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 332–344. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00039

[51] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 1157–1168.

[52] C. K. Roy and J. R. Cordy, "NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, R. L. Krikhaar, R. Lämmel, and C. Verhoef, Eds. IEEE Computer Society, 2008, pp. 172–181. [Online]. Available: https://doi.org/10.1109/ICPC.2008.41

[53] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.