Empirical Software Engineering (2025) 30:155
https://doi.org/10.1007/s10664-025-10711-4

"I see models being a whole other thing": an empirical study of pre-trained model naming conventions and a tool for enhancing naming consistency

Wenxin Jiang[1] · **Mingyu Kim**[1] · **Chingwo Cheung**[1] · **Heesoo Kim**[1] · **George K. Thiruvathukal**[2] · **James C. Davis**[1]

Accepted: 28 July 2025 / Published online: 16 August 2025

Abstract

As innovation in deep learning continues, many engineers are incorporating Pre-Trained Models (PTMs) as components in computer systems. Some PTMs are foundation models, and others are fine-tuned variations adapted to different needs. When these PTMs are named well, it facilitates model discovery and reuse. However, prior research has shown that model names are not always well chosen and can sometimes be inaccurate and misleading. The naming practices for PTM packages have not been systematically studied, which hampers engineers' ability to efficiently search for and reliably reuse these models. In this paper, we conduct the first empirical investigation of PTM naming practices in the Hugging Face PTM registry. We begin by reporting on a survey of 108 Hugging Face users, highlighting differences from traditional software package naming and presenting findings on PTM naming practices. The survey results indicate a mismatch between engineers' preferences and current practices in PTM naming. We then introduce DARA, the first automated *D*NN *AR*chitecture *A*ssessment technique designed to detect PTM naming inconsistencies. Our results demonstrate that architectural information alone is sufficient to detect these inconsistencies, achieving an accuracy of 94% in identifying model types and promising performance (over 70%) in other architectural metadata as well. We also highlight potential use cases for automated naming tools, such as model validation, PTM metadata generation and verification, and plagiarism detection. Our study provides a foundation for automating naming inconsistency detection. Finally, we envision future work focusing on automated tools for standardizing package naming, improving model selection and reuse, and strengthening the security of the PTM supply chain.

"The main idea is to treat a program as a piece of literature, addressed to human beings rather than to a computer" —D. Knuth

Communicated by: Foutse Khomh.

Extended author information available on the last page of the article

Keywords Software reuse and adaptation · Empirical software engineering · Mixed-methods study · Machine Learning (ML) · Deep Neural Networks (DNNs) · Model zoos · Package registries · Pre-trained models · Software supply chain

1 Introduction

Software reuse helps engineers develop software more efficiently and at lower costs (Griss 1993; Frakes and Kang 2005). A major form of software reuse is via software packages, which encapsulate functionality through an Application Programming Interface (API) (Decan et al. 2019; Frakes and Kang 2005) This practice is common for traditional software (Soto-Valero et al. 2019; Wittern et al. 2016; Abdalkareem et al. 2017), *e.g.,*Java packages shared via Maven[1], JavaScript packages shared via NPM[2], and Python packages shared via PyPI[3]. The same reuse practice is arising for ML-based software (Jiang et al. 2023b). In recent years, models based on deep neural networks (DNNs) have become more capable (Abiodun et al. 2018; Taye 2023) and the necessary hardware resources have become more available (Capra et al. 2020). As a result, Pre-Trained Models (PTMs), which encapsulate the capabilities of DNNs trained for specific tasks, have become increasingly popular (Jiang et al. 2022a; Han et al. 2021). PTMs are now commonly shared through PTM-specific package registries, the most prominent of which is Hugging Face (Jiang et al. 2023b). The rapid growth of PTM packages on Hugging Face, in both number and popularity, underscores their growing significance in modern software reuse practices (Jiang et al. 2024; Jones et al. 2024).

Consistent and standardized naming is a crucial element of software development and reuse (Deissenboeck and Pizka 2006; Hofmeister et al. 2017a; Alpern et al. 2024). Over the years, research has consistently highlighted the challenges posed by naming components and software package reuse (Griss 1993; He et al. 2021a; Alsuhaibani et al. 2021). In the context of PTM reuse, the selection process becomes even more complex due to the high cost of evaluating packages and the abundance of models with overlapping functionality (Jiang et al. 2023b; Taraghi et al. 2024). The naming practices and challenges for PTM packages have not been systematically studied. As shown in Table 1, the names for PTMs appear to be different in kind from the names for traditional software packages. PTM names often encode critical information about a model's architecture, size, and dataset, enabling software engineers to infer significant details directly from the name. As shown in Fig. 1, metadata in PTM packages is often embedded directly into the package, making it a critical component for effectively identifying and selecting models (Jiang et al. 2024). Therefore, in the context of PTMs, we regard metadata as an integral part of the package name. Inaccurate names and inconsistent metadata can significantly hinder searchability, reliability, and reusability.

The goal of this work is to delineate and improve the naming practices of PTMs. We focus on two themes: (1) empirical measurements of naming practices and rationales; and (2) automated identification of naming inconsistencies. In the *first theme*, our objective is

[1] See https://mvnrepository.com/.

[2] See https://www.NPMjs.com/.

[3] See https://pypi.org/.

Table 1 Top 10 package names by weekly downloads from NPM, PyPI, and Hugging Face

NPM[a] (JavaScript)	PyPI[b] (Python)	Hugging Face [c] (Pre-Trained DNN Models)
ansi-styles	boto3	ast-finetuned-audioset-10-10-0.4593
chalk	requests	chronos-t5-tiny
supports-color	urllib3	bert-base-uncased
semver	botocore	fasttext-language-identification
debug	certifi	all-MiniLM-L6-v2
has-flag	idna	clip-vit-large-patch14
color-convert	charset-normalizer	fashion-clip
color-name	setuptools	clip-vit-base-patch32
tslib	packaging	wav2vec2-large-xlsr-53-english
ms	python-dateutil	all-mpnet-base-v2

Compared to the NPM and PyPI packages (traditional software), the names of Hugging Face packages (PTMs) appear to convey more information about the package content. This paper is the first to study this phenomenon. Popularity data is as of September 2024

[a] See https://socket.dev/npm/category/popular.

[b] See https://socket.dev/pypi/category/popular.

[c] See https://huggingface.co/models?sort=downloads

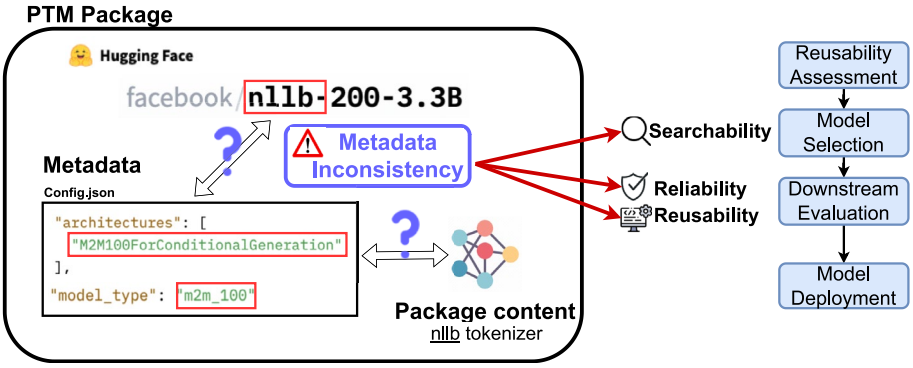

Fig. 1 Example of a PTM **metadata inconsistency** for a PTM on Hugging Face. The top part shows the PTM identifier, and the bottom part displays the PTM metadata (*i.e.,*architecture and model type) and the package content. On the right (blue boxes) is a four-step reuse process adapted from Jiang et al. (2023b). In this case, Facebook (Meta) named the model `nllb` due to the introduction of a new tokenizer, while other models on Hugging Face typically name models based on architecture changes rather than changes in tokenizers. This mismatch between the identifier and the architecture metadata impacts the new model's searchability, perceived reliability, and ultimately reusability in the PTM reuse process

to characterize PTM naming and compare it to the naming of traditional software packages. We also want to understand what elements should be included in a "good" PTM name. We take a mixed-methods approach here, combining a survey of 108 Hugging Face users with a mining study of 14,296 PTMs (1.7% of Hugging Face PTMs) from the PeaTMOSS dataset (Jiang et al. 2024). For the *second theme*, we designed and evaluated an automated tool to detect architecture-related naming inconsistencies. We experimented with various feature extraction methods, including n-gram features and advanced representations processed by CNNs and transformers. For transformer-based models, we explored continued pretraining,

fine-tuning, and contrastive learning. We evaluated each variant in our pipeline in terms of both effectiveness and efficiency.

Our findings reveal significant differences between PTM naming conventions and those of traditional software packages, highlighting unique challenges faced by engineers in the PTM domain. For the *first theme*, we report that PTM naming differs significantly from traditional software package naming. In this context, engineers follow distinct PTM naming conventions and face challenges unique to PTMs. Based on our survey data, we outline the key differences between traditional and PTM naming practices, while also presenting engineers' perspectives on these practices. A key distinction is the amount of information embedded in PTM names, which sets them apart from traditional software components. Our empirical measurements further demonstrate that the actual distribution of naming elements in practice often diverges from engineers' stated preferences. For the *second theme*, our automated tool, DARA, proves effective in detecting inconsistencies within PTM names. The best-performing architecture in terms of accuracy is the Longformer with contrastive learning, achieving up to 98% accuracy for detecting `model_type` inconsistencies and strong performance across other metadata types (`task` and `architecture`). In contrast, the N-gram + MLP classifier offers the lowest latency (0.19 ms per PTM) and highest throughput (over 28,000 PTMs/s), making it well-suited for real-time integration. These findings reveal a clear trade-off between performance and efficiency. We identified potential use cases for automated naming tools in addressing challenges related to model validation, PTM metadata generation and verification, and plagiarism detection.

To summarize our contributions:

- *Theme 1: PTM Naming Practices* We presented PTM naming practices through a survey of 108 Hugging Face users (§6.1–§6.3). We triangulated the survey study with a repository mining study on 14,296 PTM names from Hugging Face (§6.2).
- *Theme 2: Naming Inconsistencies* We developed a novel algorithm, DNN Architecture Assessment (DARA), to detect naming inconsistencies (§7). DARA combines structured architectural feature extraction with multiple classification strategies (§7.2) and achieves high accuracy with measurable efficiency trade-offs across metadata types (§7.4). We describe three use cases of DARA: model validation, metadata label generation, and plagiarism detection (§9.2).
- *Landscape of Future Work:* Our work envisions future research focused on developing automated tools for standardizing PTM package naming, supporting model selection and reuse, and enhancing the security of the PTM supply chain (§9).**Significance:** Prior work has shown that naming practices and conventions are critical in software engineering. However, this phenomenon has not been examined in the context of pre-trained deep learning models (PTMs). To fill that gap, we conducted the first empirical measurements on PTM naming conventions. Our mixed-method approach describes engineers' current practices and rationales, and uses their insights to design the first tool to identify PTM naming mismatches. Our findings provide engineers and researchers with a better understanding of PTM naming practices and offer an initial solution in this problem domain, informing the design of related tools.

2 Background and Related Work

This section presents the background information (§2.1) and reviews related work on PTM reuse and software naming conventions (§2.2).

2.1 Background

In this section, we provide related contextual information and define key terms. We cover topics including software and package reuse (§2.1.1) and how PTMs are named (§2.1.2).

2.1.1 Reuse of Pre-Trained Models

Component-based development provides a systematic approach to constructing software systems from reusable parts, significantly reducing development costs and increasing software reuse (Lau 2006; Kotonya and Lee 2014; Heineman and Councill 2001). This approach applies across various levels of software granularity, from variables and functions to larger entities such as libraries and packages, and "commercial off-the-shelf" software (COTS) (Sommerville 2015). While some components can be used directly as black-box entities with well-defined interfaces, others require significant adaptation or customization to integrate effectively within specific application contexts (Krueger 1992; Szyperski et al. 2002). Among these levels of abstraction, one of the most common modes of reuse is via software packages shared through software package registries such as NPM (JavaScript), PyPI (Python), and Maven (Java). These registries serve a combined billions of packages monthly (Wittern et al. 2016).

Similar to traditional software engineering, for deep learning approaches, engineers may use a pre-trained model with a general understanding of the task (*e.g.,*text generation) and fine-tune it for the specific application (*e.g.,*machine translation, text summarization) (Li et al. 2024; Han et al. 2021). PTM reuse is facilitated through open-source package registries like Hugging Face, PyTorch Hub, and ONNX Model Zoo, where these models are shared as reusable components (Jiang et al. 2022a). As of April 2025, Hugging Face — the largest open model registry (Jones et al. 2024) — hosts over 1.6 million PTM packages spanning domains like computer vision, natural language processing, audio, and reinforcement learning. These model registries have high industry engagement. For instance, Meta (Meta AI 2024) and Google (Google 2024a) have both published thousands of models, while DeepSeek (DeepSeek AI 2024) has recently released 14 model collections with millions of downloads. In this work, we define PTMs as packages that contain deep neural networks (DNNs), which reflects the overwhelming majority of the PTMs on the Hugging Face registry.[4]. Therefore, our approach focuses specifically on DNN-based PTM packages to ensure both methodological coherence and maximal applicability of our findings.

From a naming perspective, the key actors in the PTM supply chain (Jiang et al. 2022a) can be grouped into two roles: *name producers* and *name consumers*. Names are produced by the researchers and developers who contribute new models and assign names to the

[4] Examining the NAME dataset from REFERENCE, we observe fewer than 0.01% of all models on Hugging Face use techniques other than deep neural networks. Exceptions include graph neural networks, time series models (fewer than 60 total in the dataset, and other non-DNN models such as decision trees or random forests (typically fewer than 20 per category in the dataset)

resulting PTM packages. Names are consumed by the adopters and reengineers who reuse these models in downstream applications. Name producers may introduce novel models or innovate on existing ones through various reengineering techniques, such as transfer learning, fine-tuning, and knowledge distillation (Han et al. 2021; Davis et al. 2023; Jiang et al. 2023a). For example, reengineers often modify the model head — such as the fully connected output layer — to suit the target task (Qi et al. 2023). As PTMs grow more powerful, particularly in LLMs, reuse strategies have expanded to include zero- and few-shot learning as well as prompt tuning (Brown et al. 2020a; Radford et al. 2019; Lester et al. 2021). PTM registries like Hugging Face facilitate this reuse by allowing consumers to retrieve and deploy PTMs via model names and APIs (Jiang et al. 2023a). These consumers rely heavily on naming conventions to search, compare, and select appropriate models. Thus, clearer naming benefits all stakeholders: it improves discoverability for authors, enhances selection and evaluation for reusers, and supports better user experience for registry operators.

2.1.2 Naming Pre-Trained Models

PTM Naming Hierarchy Figure 2 presents the hierarchical naming framework in PTM registries. We define this based on our study of documentation from open-source model registries, including Hugging Face (Hugging Face 2023), ONNX Model Zoo (ONNX 2023),

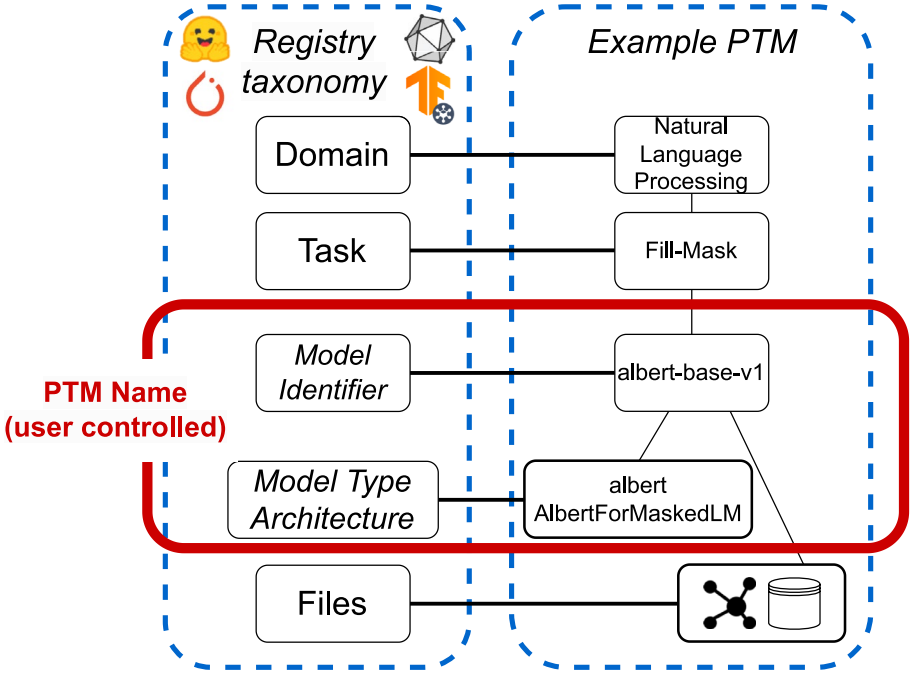

Fig. 2 Hierarchy of PTM registry naming taxonomy across representative open-source PTM registries, including Hugging Face, ONNX Model Zoo, PyTorch Hub, and TensorFlow Hub. Registries include multiple domains, such as natural language processing, computer vision, and multimodal; multiple tasks, such as depth estimation (vision), text generation (NLP); and many distinct models and model types/architectures for each task. The red box highlights the package name (model identifier and model type/architecture) — this is the focus of our study

PyTorch Hub (Pytorch 2021), and TensorFlow Hub (TensorFlow 2023). These registries generally follow similar naming conventions, including information such as the problem domain, task, model identifier, and associated files (Jiang et al. 2023c). In this work, we focus exclusively on the Hugging Face registry because (1) it is, by far, the largest PTM registry, and the only viable open-source one (Jiang et al. 2022a; 2) it provides a clear naming schema, a robust queryable API for large-scale mining, and leverages an existing structured reuse dataset (Jiang et al. 2024).

PTM registries employ a multi-tiered categorization system (Hugging Face 2023). The top level of the hierarchy represents the model's domain, which refers to the broad area of application where a group of models is utilized, such as Natural Language Processing or Computer Vision (Islam et al. 2023). Within a domain, *tasks* are specific problems that models aim to solve, like Sentiment Analysis or Named Entity Recognition in the Natural Language Understanding domain. A *PTM* is an instance of a trained model for a specific task, complete with weights, parameters, and configuration metadata. A PTM package's `config.json` file details the model's *architecture* and specifies the *model type* and the task it is designed for, *e.g.,* `Albert` for the model type and `QuestionAnswering` for the task. ***Definition 1*** summarizes these terms:

Definition 1: Naming Hierarchy in PTM Registries

- **Domain and task**: High-level metadata that are part of the *model type*.
- ***PTM name***: Defined as the combination of a package *identifier* (*e.g.,*`albert-base-v2`) and the *model type* or *architecture* indicated in the metadata (*e.g.,*`AlbertForQuestionAnswering`).

In the Hugging Face registry, we consider both the model metadata and its files to be objective factors because they are clearly defined by the model's architecture, dataset, and performance metrics, which are based on measurable attributes and standardized benchmarks. On the other hand, the package name is a subjective element, as it is manually crafted by PTM contributors based on their own interpretation of the model's purpose, function, and approach, among other considerations. This subjectivity arises from the flexibility contributors have in deciding how to represent key aspects of the model, allowing them to incorporate personal interpretation and preferences and leading to variations in naming conventions.

This subjectivity makes the naming process a critical point where inconsistencies or ambiguities can occur. According to prior research (Jiang et al. 2023b), the engineers involved in the PTM ecosystem commonly rely on either package identifiers or architectural labels when searching for PTM packages appropriate for their specific tasks. The subjective components of a package involve judgment, both for the namer and for the potential re-user, which impacts how reengineers and adopters discover and assess a PTM.

PTM Naming Elements and Conventions Two key concepts in PTM naming practices are naming *elements* and how they are combined to make naming *conventions*.

In the context of PTM package naming, **naming elements** are essential components that make up the name of a model. For example, a model might have naming elements structured as "Architecture-Size-Dataset Characteristic" (`deberta-base-uncased`), where

the name includes information about the model's architecture (`deberta`), size (`base`), and dataset characteristic (`uncased`). ***Definition 2*** formalizes this notion.

Definition 2: PTM Naming Elements

Naming elements: The elements that comprise a model's name, representing different categories or attributes that are combined to form the full model name.

Another critical aspect of PTM naming practices is the **naming convention**, which refers to the norms and rules governing what elements are included in a package identifier. For example, in a model named `deberta-base-uncased`, the naming elements reflect the implementation details of the model. In contrast, a name like `fasttext-language-identification` includes both the *implementation* unit (*i.e.,*the FastText model) and the *application* goal (*i.e.,*language identification). We define naming conventions for packages in ***Definition 3***.

Definition 3: PTM Naming Convention

Naming convention: Norms and rules that governs how elements are named and combined to form a package name, ensuring a clear link between the implementation details and the application's objectives (Lawrie et al. 2006).

2.2 Related Work

We first present a broad review of known challenges in the PTM reuse process (§2.2.1), and then focus on prior work on naming in traditional software (§2.2.2). Prior work directly on PTM names is included in our problem statement (§3).

2.2.1 Challenges in the PTM Reuse Process

Reusing PTMs can speed up development, but it also brings unique challenges that differ from traditional software reuse. A common approach is fork-based reuse—cloning a model and modifying its architecture or retraining it to fit specific needs (Jiang et al. 2023a; Bhatia et al. 2023). However, this method complicates conceptual reuse, adaptation, and deployment (Davis et al. 2023). Reusing PTMs presents several challenges, including missing attributes, inconsistent metadata, and security or privacy risks (Jiang et al. 2023b). In the same study, Jiang et al. reported that many experienced Hugging Face users expressed concerns over model trustworthiness due to inconsistent or missing metadata, including names. Other studies highlight challenges in analyzing PTM vulnerabilities (Kathikar et al. 2023) and managing their lifecycle during reuse (Castaño et al. 2023). Taraghi et al. conducted a qualitative analysis of Hugging Face forum discussions and found that understanding and effectively reusing PTMs remains the most significant challenge (Taraghi et al. 2024). These issues underscore the need for standardized, descriptive naming conventions to support more effective PTM management, which can further enhance discoverability and reliability, and reduce both computational and financial costs in the reuse process (Sens et al. 2024; Schelter et al. 2018). Our work presents the first empirical study on PTM package naming practices and provides insights into standardized naming conventions to ensure consistency and reliability.

2.2.2 Software Naming Conventions

The naming of software components is crucial for code readability, maintainability, and collaboration (Seacord et al. 1998; Lawrie et al. 2007; Butler et al. 2010; Hofmeister et al. 2017a). Existing studies have primarily concentrated on the naming of variables (Hofmeister et al. 2017a), functions (Alsuhaibani et al. 2021), and classes (Butler et al. 2011). Researchers have documented two common strategies for naming:

by the implementation approach, *i.e.,*"*how it works*"; and by the application goal, *i.e.,*"*what it does*" (Henninger 1994; Hofmeister et al. 2017a). These strategies can be reflected in the package's naming elements (Definition 2) and the resulting naming convention (Definition 3), although the specific strategies used for PTMs have not previously been described.

While naming strategies have been widely studied for fine-grained implementation elements such as variables and functions, the naming conventions for software packages have received less attention. It is known that clear and descriptive names enhance the discoverability and reuse of packages (Deissenboeck and Pizka 2006; Lawrie et al. 2006; Abdellatif et al. 2020; Alsuhaibani et al. 2021). However, there has been no systematic work linking naming conventions directly to software reusability and reliability at the package level. Our study is the first to explore this connection, with a particular focus on the reuse of PTM packages.

Well-chosen names allow users to quickly grasp a package's purpose, aiding in the selection process. As an illustration, Table 1 shows popular packages from NPM (JavaScript), PyPI (Python), and Hugging Face (PTMs). Anecdotally, looking at Table 1, we see an interesting phenomenon: the names of PTMs often convey more information than those of traditional software packages.

- In **traditional software package registries** like NPM, package names often follow a convention of indicating "what it does", typically focusing on the package's main feature or purpose, which helps developers easily search for and select packages based on their specific needs (NPM 2023; Guido van Rossum et al. 2001). For example, Python's PEP8 style guide recommends that package names be short, descriptive, and easy to understand, while avoiding unnecessary complexity (Guido van Rossum et al. 2001). This approach is also evident in names like `color-convert` and `color-name`,which clearly signal their color-related functionality. In contrast, a name like chalk is less clearly associated with color. This name has not kept chalk from being used — it is one of the most popular packages on NPM — but it does add a barrier for understandability.
- In contrast, we observe that open-source **PTM registries** like Hugging Face, PyTorch Hub, and ONNX Model Zoo tend to follow a more detailed naming convention. These names are usually longer compared to traditional package names frequently embed important metadata about the model, such as its architecture, training dataset, or performance metrics, offering users a more comprehensive understanding of the model's capabilities and purpose (Jiang et al. 2022a, 2023b; Di Sipio et al. 2024). For example, in the Hugging Face column of Table 1, the PTM named `ast-finetuned-audioset-10-10-0.4593` is an ast (*i.e.,* Audio Spectrogram Transformer) model fine-tuned on the `AudioSet` dataset, achieving performance (mean Average Precision) of

0.4593. The two `10` s represent the frequency and time strides, respectively. Similarly, the PTM named `bert-base-uncased` is a base version of the BERT model without case sensitivity. Model names like `fasttext-language-identification` convey information about the task and application, helping users make informed decisions when selecting models (Han et al. 2021).

3 Problem Statement

Recently, there has been increased attention paid to the broader challenges associated with PTM naming. Prior research has highlighted difficulties in PTM reuse, specifically related to model selection and management (Tan et al. 2024; Taraghi et al. 2024). Addressing these challenges requires more consistent naming practices and enhanced transparency of PTMs (Di Sipio et al. 2024). Furthermore, industry stakeholders, including Google and HiddenLayer, have emphasized the importance of maintaining accurate inventories of PTMs, tracking their provenance to enable rapid responses to emerging vulnerabilities (Hepworth et al. 2024; HiddenLayer 2024).

An illustrative example of a PTM **naming inconsistency** is presented in Fig. 1. While the appropriateness of a PTM name can be subjective, the PTM ecosystem currently lacks standardized guidelines to inform and evaluate naming practices, making model selection and downstream reuse more difficult. For example:

- NVIDIA engineers made a typo in the model card of their speech recognition model `Parakeet` (with 15K monthly downloads). The model_config incorrectly specified `EncDecCTCModel` instead of `EncDecRNNTModel`. This inconsistent architecture metadata caused users significant debugging effort and confusion when attempting to load the model properly (NVIDIA 2024).
- Similarly, Longformer initially listed `BartModel` as its base architecture, which confused users who expected to load it using `LongformerModel`. The issue was later resolved when Hugging Face updated the configuration to correctly reference the `LongformerModel` class (Peng 2021).
- The LLaVA model also suffers from naming inconsistencies. For instance, users have expressed confusion over similarly named models such as `LLaVA-Plus`, `LLaVA-1.6`, and `LLaVA-Next`, with no clear guidance on which is the latest or performs best overall (shersoni610 and Li 2024).

These naming inconsistencies can negatively impact both the *searchability* of models, making them harder to discover, and their *reliability*, as misleading names may confuse the model's true functionality. We discuss these two topics in more detail next.

Definition 4: PTM Naming Inconsistency

PTM Naming Inconsistency: A deviation where the name (*i.e.,* identifier and architectural metadata, *cf.* Definition 1) of a PTM package does not correctly reflect its underlying architecture or intended functionality, leading to potential confusion or misinterpretation by users.

Impact on Package Searchability Model names play a crucial role in determining search capabilities within platforms like Hugging Face, as they serve as a primary source for generating metadata. However, user-generated metadata often contains errors, forcing engineers to manually inspect models to verify their suitability (Jiang et al. 2022b, 2023b). For instance, mislabeling "`ResNet` " as "`VGGNet` " is a clear naming mistake beyond mere subjectivity. Missing details and inconsistencies in names can create significant challenges in PTM reuse, leading to difficulties in understanding model capabilities, maintaining code, and collaborating effectively (Butler et al. 2010).

The lack of standardized metadata and clear documentation significantly complicates the discovery, evaluation, and reuse of PTMs (Jones et al. 2024; Jiang et al. 2024). According to prior research (Taraghi et al. 2024; Jiang et al. 2023b), engineers often depend on PTM package names as primary indicators of a model's architecture and functionality, particularly when model cards (*i.e.,* `README`s) and metadata are frequently insufficient. Inconsistent or misleading names can lead to the selection of unsuitable or incompatible models, resulting in wasted time and effort during the development process.

One recurring challenge in PTM reuse is the inefficiency caused by unclear or inconsistent naming conventions (Tan et al. 2024; Jiang et al. 2023b; Taraghi et al. 2024). Well-structured naming practices enhance transparency, streamline searchability, and reduce the need for manual verification by ensuring that names accurately convey essential information such as model architecture, size, and dataset characteristics (Seacord et al. 1998; Deissenboeck and Pizka 2006). For example, Fig. 1 illustrates a PTM metadata inconsistency where there is a mismatch between the model identifier (*e.g.,*"`nllb` "), package content (*e.g.,*"`nllb` " tokenizer) and its architecture metadata (*e.g.,*"`m2m` "). While the appropriateness of PTM names can be subjective, the absence of standardized guidelines within the PTM ecosystem complicates the evaluation of naming practices. Our study examines the methods practitioners use to identify such inconsistencies and introduces an automated tool designed to detect incorrect model architecture and types.

To address these challenges, researchers have proposed solutions like model nutrition labels and queryable model zoos, which aim to provide comprehensive, standardized metadata about models, including their intended use, limitations, and performance metrics (Chmielinski et al. 2022; Li et al. 2022; Tsay et al. 2020). However, recent studies indicate that these approaches have not been fully adopted in practice, and many PTM models still lack key attributes and metadata (Jiang et al. 2024, 2023b). Consequently, engineers often rely heavily on PTM names to make initial decisions and communicate about models. Our study optimizes PTM naming practices to improve searchability by identifying the information that engineers want to be conveyed in names, and we provide an automated tool to detect some kinds of naming defects (inconsistencies between model architecture and metadata), further enhancing model selection.

Impact on Package Reliability and Security In addition to searchability, accurate naming in PTM packages is also critical for ensuring reliability and security. Prior research on traditional software packages highlights risks such as typosquatting and package confusion, where misleading or poorly chosen names can result in the unintentional download of malicious content (Taylor et al. 2020a; Neupane et al. 2023). Similarly, PTM packages with unclear or inconsistent names can obscure vulnerabilities like backdoors, introducing

significant risks to downstream applications (Kathikar et al. 2023; Wang et al. 2022; Gu et al. 2019). The rapid proliferation of new models and the complexity of PTM architectures make clear and consistent naming essential to prevent these security threats. Accurate names that align with the package's content, architecture, and metadata can mitigate these risks and improve overall package reliability.

Manually verifying naming consistency in PTM reuse is not only time-consuming but also prone to errors. In the context of PTM integration, models are often incorporated with minimal inspection, making reliance on manual checks inefficient and risky. This can lead to the oversight of critical issues such as mislabeled models or hidden backdoors, which pose security threats. Although automated PTM reuse, facilitated by task-based assessments, is becoming more feasible, these tools still require engineers to pre-select models based on names and metadata (You et al. 2021a; Zhang et al. 2024). This dependency underscores the need for clear and consistent naming conventions, as they directly influence the accuracy and security of automated reuse processes. Prior research has shown that secure software reuse and effective searchability are significantly enhanced by standardized and validated metadata practices (Khalid and Zimányi 2024; González and Van Der Meer 2004). Our work provides an automated way to verify metadata consistency, which can reduce manual effort and improve the security and reliability of PTM reuse.

Summary of contribution in light of prior work

Prior work shows naming conventions are crucial for the software reuse process, including PTM packages. PTM packages are known to have distinct naming challenges, but the details of those challenges have not been elaborated, nor have tools been developed to mitigate them. Our work is the first empirical study on PTM naming practices and includes the development of an automated tool to address metadata inconsistency, with a vision for future improvements. By promoting more **accurate and standardized naming conventions**, our study aims to enhance (1) the management and validation of PTM packages in registries like Hugging Face, improving organization and discoverability, and (2) the user experience by streamlining model search, reducing manual verification, and minimizing engineering overhead.

4 Research Questions and Justification

4.1 Research Questions in Light of the Previous Literature

To address this gap and promote more accurate and standardized PTM naming conventions, we present the first study in this area. Our research focuses on two themes: (1) empirical measurements of naming practices to enhance standardization, and (2) automated identification of naming inconsistencies to improve naming accuracy. We ask:

Theme 1: Measuring Naming Practices

RQ1 How is PTM naming different from traditional software package naming?

RQ2 What elements should be included in a PTM identifier?

Theme 2: Detecting Naming Inconsistencies**RQ3** How do engineers identify naming inconsistencies?

RQ4 How can naming inconsistencies be detected automatically? Specifically, how well can architectural metadata inconsistencies be detected?

We note that RQ1-3 primarily focus on the PTM package identifier, while RQ4 shifts attention to the architectural metadata which is defined as part of PTM name (*cf.* Definition 1). Although this may appear as a shift in terminology – from naming practices to architectural metadata – the scope remains consistent with our definition of PTM name as a combination of both elements. RQ4 specifically investigates whether the architectural metadata aligns with the model's actual behavior, which we consider a critical aspect of naming reliability and transparency in the ecosystem.

4.2 How are the Answers Expected to Advance the State of Knowledge?

Our study has both *theoretical* and *practical* motivations that contribute to advancing the state of knowledge in software engineering and machine learning.

Theoretically, understanding developer behavior, particularly in the context of software naming practices, is a fundamental area of software engineering research. Such software engineering-theoretic knowledge holds intrinsic value, even if it does not lead to immediate applications. Our work extends prior research on naming practices, such as Feitelson's studies on code-level naming (Feitelson et al. 2020; Alpern et al. 2024), by focusing on package-level naming within the pre-trained model (PTM) ecosystem. The research literature is silent on the naming of higher-order entities such as software packages, and no comparison has been performed between the naming of traditional packages and of PTM packages. Addressing this gap, our research questions (RQ1) and subsequent analyses (RQ2-3) aim to provide novel insights into how naming conventions evolve in these distinct contexts.

Practically, our work was originally motivated by the findings of Jiang et al. (2023b), one of the first qualitative studies on PTM reuse from the software engineering perspective. Their results highlighted that engineers often depend on model names for key information when searching for and discovering models, especially when metadata and documentation are insufficient. Our research specifically addresses three major issues within the PTM ecosystem:

- *Transparency, discoverability, and reusability*: Previous work has shown that the documentation available for Hugging Face models is often inadequate, leading to reduced transparency, discoverability, and reusability of PTM packages (Taraghi et al. 2024; Jiang et al. 2023b). Since model names and metadata are primary sources of information for users (Di Sipio et al. 2024; Hepworth et al. 2024), inconsistencies or omissions in these identifiers can hinder effective reuse. The alternative—manually inspecting model architectures—is more resource-intensive and time-consuming. Our research offers an automated approach to detecting mismatches and discrepancies in model metadata, a critical aspect of PTM naming, to enhance transparency, discoverability, and reusability of PTM packages. This represents an initial solution to the problem, we do not claim that it fully resolves all challenges related to metadata inconsistencies.
- *Adversarial attacks on PTMs*: Architectural adversarial attacks, such as model backdoors, are closely tied to inaccuracies in model metadata. For example, an attacker

could embed a backdoor within a model that triggers incorrect outputs for specific inputs (Bober-Irizar et al. 2023; Langford et al. 2024). Our work has the potential to improve the accuracy and standardization of PTM package naming conventions, thereby reducing risks associated with potential backdoors and model plagiarism by identifying discrepancies. This, in turn, facilitates a more reliable and efficient PTM package reuse process.

– *Supply chain attacks*: Supply chain attacks, such as typosquatting, are another threat to PTM reliability. In such cases, a model might claim to be a well-known architecture like Llama but actually be a different model, potentially embedding malicious code. Our approach detects inconsistencies by analyzing the architecture and generating more accurate metadata labels, thereby improving efficiency and reducing risks in model selection and validation.

Theme 1 describes PTM package naming practices in Hugging Face, and Theme 2 develops practical methods for detecting naming inconsistencies. Practitioner interest, as revealed through our survey, underscores the relevance of this work, particularly in establishing naming conventions and standards within the PTM ecosystem. Our findings contribute to improving naming practices and addressing reuse challenges, laying the foundation for more efficient and secure PTM reuse.

4.3 Structure of the Rest of this Paper

We answer RQ1-3 through a survey and mining study. We follow conventional empirical software engineering research methods for these questions. Section 5 describes our methodology and §6 shares our results for these questions.

Our goal in RQ4 is to develop an automated system to detect architectural metadata inconsistencies. In §7, we describe the requirements, our system design, and the results of our evaluation.

The remainder of the paper describes the threats to the validity of our work (§8), discusses the implications of our findings for engineers and future research (§9), and concludes (§10).

5 Methodology for Research Questio 1-3

This section presents our methodology to answer Research Questions 1, 2, and 3. Because these questions involve both understanding developer perceptions (RQ1-3) and analyzing naming practices at scale (RQ2), we apply a mixed-method approach (Storey et al. 2025; Johnson and Onwuegbuzie 2004) using typical empirical software engineering methods. We conducted a survey study (§5.1), and complemented it with a repository mining study on Hugging Face (§5.2).

Figure 3 shows the relationship between RQs and methods. Because RQ4 applies a system design methodology, we defer its answer to §7.

5.1 Method for Survey Study (used for RQs 1-3)

This section presents our methods for the survey portion of our investigation of RQs 1-3: preliminary analysis (§5.1.1), instrument design (§5.1.2), recruitment (§5.1.3), and analysis (§5.1.4).

5.1.1 Preliminary Analysis

We conducted a preliminary analysis on 200 Hugging Face models stratified by domains to build an initial understanding of the naming conventions in PTMs packages. Based on those results, we identified 12 naming elements (Table 2) and 4 naming conventions (Table 3). We expanded our preliminary analysis to an additional 50 models and observed no emergence of new categories. These categories informed the design of our instrument, ensuring that we captured the key naming elements and conventions prevalent in the domain while providing clear definitions that align with engineers' familiar terminology.

As described in §2, in Hugging Face, models are categorized by (Fig. 2) the overarching domain and the tasks within them. To control for potential variability along these dimensions, we collected a stratified random sample along task types (*e.g.,*depth estimation, question answering) within each major Hugging Face domain (*e.g.,*Computer Vision, Natural Language Processing, and Multimodal) to ensure coverage of diverse modeling goals and application areas. This method allows for representation across different model categories and domains, reducing the potential for selection bias. Given the size and diversity of the total model population on Hugging Face (approximately 974K PTMs as of September 2024), our sample size provides a 99% confidence level with an 8% margin of error, indicating strong statistical reliability. These results lead us to conclude that the identified naming conventions and elements are comprehensive, as no new patterns emerged in the expanded sample. However, the practical distribution of these elements and engineers' preferences remain unclear.

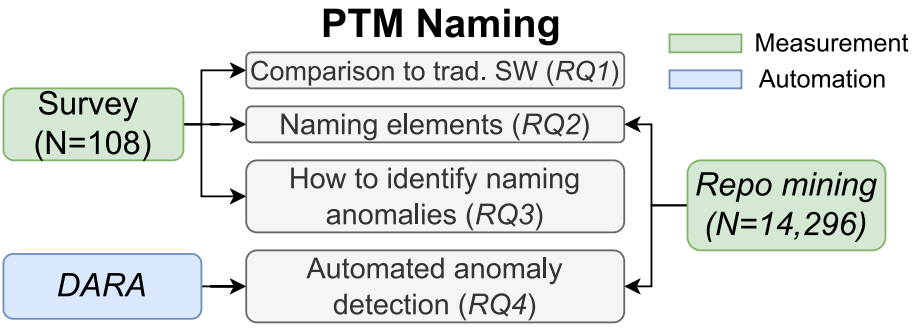

Fig. 3 Relationship between research questions (RQs) and methodology. We conducted a survey study to address RQ1-3 and developed an automated tool, DARA, to address RQ4. Additionally, a repository mining study was conducted to support answers for RQ2 and RQ4

Table 2 Definition and examples of PTM naming components

Naming component	Definition	Example
Architecture [A]	The foundational structure or design of the deep learning model.	bert, albert, resnet
Model size [S]	The model complexity *e.g.,*# of parameters or architectural units.	50, 101, base, large, xxlarge
Dataset [D]	The specific dataset used for training or evaluation of the model.	squad, imagenet
Dataset characteristics [C]	The traits of the dataset, *e.g.,*text casing, image dimensions.	case, uncased, 1024-1024, 244
Model versioning [V]	The version of the model.	v1, v2
Language [L]	The human language for which the model is trained on.	english, chinese, arabic
Task [T]	The problem or functionality the model is designed to address.	qa, cls, face-recognition
Training process [R]	The algorithm or approach used for training the model.	pretrain, sparse
Reuse method [F]	The approach employed to adapt a PTM for downstream tasks.	fine-tune, distill, few-shot, instruct
Number of layers [Y]	The depth of the neural network in terms of layers.	L-12, L-24
Number of parameters [P]	The total number of trainable parameters in the model.	100M, 8B
Other [O]	A portion of the model name not classified above categories.	demo, test

All components were identified from our preliminary study of 200 PTM packages, with no additional occurrences observed in the subsequent analysis of 50 packages. The boldfaced components were specifically identified in our preliminary study's second half (*i.e.,*the last 100). An example real name is `bert-base-uncased-finetuned-spam`. which is of the form `A-S-C-F-D`. Another example is `Meta-Llama-3.1-8B-Instruct`, which is of the form `O-A-V-P-F`

Table 3 Definition and examples of PTM naming conventions

Naming conventions	Definition	Example
Implementation Unit	The detailed configuration and design aspects of a model that shape its structural and operational framework.	bert-base-uncased, whisper-large-v2-pt-v3
Application/Task	The real-world applications or specific tasks that the models are designed for.	fake-news-detector, question-answering
Implementation with App./Task	The model name that includes both the implementation unit and the application/task it is designed for.	distilroberta-base-finetuned-fake-news-detection
Other	The model name that does not fall into non of the above categories or have unclear meaning.	potat1, csproject

These conventions were identified from our preliminary study of 200 PTM packages

5.1.2 Survey Instrument Design

We developed our survey instrument following Kitchenham and Pfleeger's guidelines for survey design (Kitchenham and Pfleeger 2008). They recommend six steps. In this section we cover the first four – defining objectives, designing the survey, developing the instru-

ment, and survey evaluation. The remaining two steps – data collection and analysis – are discussed in §5.1.3 and §5.1.4, respectively.

Instrument Development Step 1: Search the relevant literature: We began our study with a comprehensive literature review focused on naming practices within software engineering. We conducted the literature review using a keyword search method on the DBLP database.[5] The goal of this review was to inform the survey design by identifying existing naming practices and reuse challenges relevant to PTMs. To identify papers on software naming practices, we searched using the following Boolean keyword query:

> (*"software name"* **OR** *"naming convention"* **OR** *"naming practices"*) **AND** (*"software"* **OR** *"programming"* **OR** *"software package"*)
>
> We also included related terms like "empirical study" and "software naming" to broaden the scope. Separately, to identify relevant work on PTM reuse, we used the following query:(*"pre-trained model"* **OR** *"machine learning model"* **OR** *"AI model"* **OR** *"deep learning model"*) **AND** *"reuse"*

These search queries resulted in 22 papers on software package naming (*e.g.,*Seacord et al. 1998; Lawrie et al. 2007; Butler et al. 2010) and 2 papers on PTM reuse (Jiang et al. 2023b, 2022a) at the time of designing our instrument. We excluded 16 papers that primarily focused on naming at the method or class level, which were less relevant to our focus on package-level and model-level naming conventions (*e.g.,*Allamanis et al. 2015; Wang et al. 2023b; Butler et al. 2010).

The remaining 8 papers were analyzed in depth by the lead author, who was responsible for survey and instrument design. Through multiple close readings and reflection, the author synthesized key themes on naming motivations and reuse challenges in traditional software and considered how they may transfer to the PTM ecosystem. This thematic synthesis set the objectives of our study and informed the development of our research questions (§4).

Step 2: Construct an instrument: The initial instrument includes five sections: demographic questions, comparison between PTM naming and traditional software engineering (RQ1), PTM naming elements and conventions (RQ2), and practices on naming inconsistency detection (RQ3).[6] The survey was designed to take approximately 15 minutes and primarily consisted of multiple-choice questions, with a few Likert-scale and open-ended questions included to gather more detailed insights from participants.

Step 3: Evaluate the instrument: To ensure the survey instrument was appropriate and aligned with our research goals, we conducted a pilot study in two rounds. Our target population consists of practitioners and researchers familiar with PTMs, particularly those who have used or published models on Hugging Face.

In *Round 1 (internal pilot)*, we invited three PhD students conducting ML research with prior experience using PTMs from Hugging Face. They completed the survey and then

[5] See https://dblp.org/.

[6] For completeness, we note that the survey instrument also collected data on software engineers' use of PTM interoperability tools, such as the Open Neural Network Exchange (ONNX), with results reported in Jajal et al. (2024). Those questions were presented on a separate page after all questions related to PTM naming, so we do not anticipate any priming effects or interactions between the two parts of the survey.

participated in interviews focused on (1) whether the survey comprehensively covered relevant naming and reuse issues, and (2) identifying confusing, redundant, or unnecessary questions. After this round, the survey was revised to clarify wording, increase conciseness, and align it with the terminology and activities of users on Hugging Face. We reviewed the changes with these participants to confirm the changes addressed their concerns.

In *Round 2 (external pilot)*, we distributed the survey to 30 Hugging Face PRO users and 100 ORG users. However, the response rate was very low (1%). In response, we substantially simplified the survey. We consolidated numerous questions and restructured the survey. For example, we removed several open-ended questions, eliminated all Likert-type questions (which required significant time and effort from pilot participants), and converted as many questions as possible into structured multiple-choice questions (based in part on the kinds of responses received in the pilots). Retesting with the 3 subjects from the internal pilot, the revised survey took 5–8 minutes to complete and still produced data relevant to our research questions. We allowed respondents to leave answers blank to improve the response rate, which reduced the volume of responses per question but increased the total number of responses.

Step 4: Document the instrument: Table 4 presents example questions from the final survey used in this study, which includes demographic questions and queries on PTM naming practices. To encourage higher response rates, all questions were optional. Additionally, the survey provided a definition of PTM naming (§2.1) for clarity. The final instrument is available in Data Availability. Our study was approved by institutional IRB #2022-606.

5.1.3 Recruitment

We compiled email addresses from Hugging Face's PRO and ORG (*i.e.,* organization) user accounts.[7] PRO accounts mean that the users actively pay for the features provided by Hugging Face. To filter for practitioners, we collected users from organizations which were labeled as `company`, `community`, and `non-profit`, excluding `university`, `classroom`, and no labeled organizations.

Our survey aimed to achieve a 95% confidence level with a margin of error of 10%. Previous statistics indicate that Hugging Face had over 1.2 million users by 2022 (Garg 2023), and it kept increasing in recent years. Therefore, we needed at least 96 participants to ensure statistical significance (Sur 2021).

To ensure we targeted the appropriate population (§5.1.2), we randomly shuffled the collected PRO and ORG users. We then distributed the survey in batches of 100 emails until reaching our desired sample size. In total, we distributed our survey to 228 PRO users and 1,985 organization members. As an incentive, each participant was compensated with a $10 gift card. To ensure the quality of our survey responses, we began the survey with a screening question: *"Do you regularly interact with a model exchange platform such as Hugging Face, PyTorch Hub, or an internal registry?"*. We also included a CAPTCHA to verify that responses were submitted by humans. 119 users responded to our survey, resulting in

[7] In our judgment, this does not violate Hugging Face 's terms of service (Hugging Face, Inc. 2024). These terms of service permit usage under strict compliance with applicable laws and regulations. Our study, being controlled and approved by our institutional IRB, is in compliance with these terms. We used sampling to ensure our activities are neither excessive nor disruptive and to respect Hugging Face's data privacy and user rights.

Table 4 Example survey questions

Topic	Example questions
(1) Demographics	(1) What is the size of your organization?
	(2) What deployment contexts do you work on?
	(3) How many PTM packages have you used/created from model registries?
(2) Comparison to Traditional Software	(1) How is PTM naming similar/different from naming traditional software packages such as those on NPM or PyPi? Why do you think that is?
(3) Naming practices	(1) Which naming convention do you prefer when reusing PTMs from model registries like Hugging Face?
	(2) Here is a list of PTM naming elements. Check each box if you think it would be important to include that element in a name.
	(3) Machine learning models are often adapted for improved performance or specific needs. These modifications might involve changing the architecture, training regime, and dataset. What kinds of modifications might necessitate a new model type/architecture for the model (*e.g.,*"distilBERT"), as opposed to continuing to hyphenate the name (*e.g.,*"albert-v2-50M")?
(4) Naming challenges	(1) In your experience, do the PTMs available in model registries accurately describe their behavior/content? What discrepancies have you experienced? Please explain.
	(2) In your regular practice, do you think you would notice if a PTM had an incorrect name? Check the box if you think YOU WOULD NOTICE if the naming element were incorrect.
	(3) If you think you would typically notice one or more of these naming elements being incorrect, please tell us what process you follow to do so (e.g., reading model card, reading source, etc.).

The survey had 13 questions in total. The survey used a mix of multiple-choice questions (Topics 1 and 2), checkbox questions (Topics 1 and 3), and open-ended responses (Topic 2 and 4). The full instrument is available in the supplemental material (Data Availability)

a response rate of 5.4%. Of these, 108 actively interacted with Hugging Face, which we consider as high-quality responses. We used this subset in our analysis, yielding an effective response rate of 4.9%. Our recruitment strategy achieved the desired statistical significance at a 95% confidence level by specifically targeting experienced users with in-depth insights into PTM naming conventions. However, we acknowledge that this focus on experienced users may introduce biased responses. We discuss this further in §8. Table 5 gives subject demographic information.

A key consideration in our analysis was whether to differentiate between respondents who only use PTMs and those who also create them. In our study, all respondents have experience using PTMs, but only 34 out of 108 (31.5%) have not created a PTM. As a result, approximately 70% of respondents have experience both using and creating PTMs. Given the small size of the non-creator group, making statistical distinctions between these two groups would be infeasible.

5.1.4 Survey Analysis

The first step in survey analysis is to filter low-quality responses. We determined that it was unnecessary to implement techniques for filtering low-quality or erroneous responses, such as attention checks, knowledge checks, or manual reviews. Since we specifically targeted Hugging Face users with direct experience in reusing PTMs, the responses inherently met

Table 5 Participant demographics (N=108 but some left blank)

Type	Break-down
ML Exp.	<1 yr. (11), 1-2 yr. (24), 3-5 yr. (32), >5 yr. (36)
SE Exp.	<1 yr. (6), 1-2 yr. (12), 3-5 yr. (22), >5 yr. (63)
Org. Size	Small (1-50 employees, 58), Medium (51-250 employees, 19), Large (over 250 employees, 26)
Deployment Env.	Web application (68), Desktop application (21), Cloud and data center (61), IoT/embedded systems (12), Mobile (15), Other (6)
# of PTMs used	0 (0), 1-5 (38), 6-10 (18), 11-20 (19), >20 (28)
# of PTMs created	0 (34), 1-5 (37), 6-10 (15), 11-20 (5), >20 (12)

The majority of participants have intermediate or expert experience in both ML and SE. All participants are PTM users

our quality standards. However, to ensure accuracy, we manually inspected all data to confirm the consistency of the results and validate our assumptions about response quality.

The majority of our survey questions were formulated as multiple-choice/checkbox, which are simple to analyze. Two open-ended textual responses required subjective analysis. We analyzed these as follows:

1. The first open-ended question, for RQ1, asked: *"How is PTM naming similar/different from naming traditional software packages such as those on NPM or PyPI? Why do you think that is?"*. Responses were lengthy and variable, so two analysts worked together to analyze them following thematic coding methods (Guest et al. 2011).

   - First, the analysts independently performed memoing (Saldaña 2021) to develop initial themes. They initially agreed on ∼65% of codes.
   - Next, they met to iteratively refine the themes (initial codebook), assessing their reliability, merging entangled codes, and discarding rare ones.
   - After a second coding round, they met to resolve disputes and adjust codes until consensus was reached. For instance, "CONFIG" and "REUSE" were merged due to overlap and unclear definitions. After refining the themes, they reviewed all previous codes and achieved unanimous agreement.

2. The second open-ended question, for RQ3, asked respondents to describe their practices for identifying naming inconsistencies. Two analysts reviewed the data and determined that the analysis was more straightforward and showed less variability than the first question. As a result, thematic coding methods were deemed unnecessary, and the analysis was conducted by a single researcher for efficiency.

5.2 Method for Repository Mining in Hugging Face (used for RQ2)

The survey data permitted us to answer RQ1 and RQ3. For RQ2, we complemented the survey with a repository mining study. This section details the methods of our mining study (§5.1) which offers a more comprehensive understanding of practical PTM naming conventions and evaluates whether these practices align with engineers' stated preferences.

5.2.1 Data Collection

We use the latest PTM dataset named PeaTMOSS (Jiang et al. 2024), which was collected in August 2023. In our study, we focused on the identifiers of PTM packages from PeaTMOSS dataset which has over 50 downloads. This included 14,296 PTMs (5% of the full dataset), representing the PTM packages actively used by engineers. We excluded models with low download counts because (by definition) they do not reflect common usage. Similar to the long tail in traditional software package registries, such models are often experimental, student work, or throwaways. Lower usage may also be correlated with lower model quality, which may be associated with poorer naming practices. However, we acknowledge that this exclusion may introduce potential bias, as discussed in §8.

5.2.2 Repository Mining Pipeline Design Using LLM

The aim of our repository mining study is to extract naming elements and conventions from open-source PTM package names. Since these names often reflect nuanced, context-specific choices by engineers, the pipeline requires strong natural language understanding. Large language models have demonstrated strong capabilities in this area (Bubeck et al. 2023). To analyze naming practices at scale, we designed a rule-guided prompt engineering approach using the ChatGPT API (`gpt-4-turbo`). We chose to apply LLM technology to this language processing task because of its ability to parse diverse naming patterns, infer implicit conventions, and handle edge cases that are difficult to capture with regular expressions or static heuristics. Our pipeline was then built around this capability, with a focus on interpretability and reproducibility through clear task instructions and output formatting constraints.

Figure 4 illustrates the LLM pipeline we developed. In this process, we input the name of a PTM package and employ two distinct prompts (§5.2.3) to deduce the naming conventions and elements associated with that name.

Following the methodology suggested by Zamfirescu-Pereira et al. (2023), we utilized a structured process to design the prompt for our measurements. For each measurement, we first crafted a textual input (*i.e.,*prompt) and conducted a preliminary analysis on 20 randomly selected models. *First*, we incorporated the definitions from our survey into the prompt to clarify naming elements and conventions. *Then*, we enriched the prompt with definitions and examples for each specified component to unify semantic naming patterns. *Finally*, we added several rules to regulate the output format, and apply few-shot prompting, presenting example inputs alongside desired outputs (Brown et al. 2020b).

This process involved iterative refinement focused on two primary objectives by adjusting the *output format instructions* and refining the *definitions* used in each prompt: (1) Ensuring the generated output adheres to a well-defined, easily parsable format; (2) Optimizing the performance of the output, evaluated through manual inspection.

Figure 4 presents the structure and some content of each prompt. We also briefly describe each prompt as follows:

– **Naming Elements Prompt:** The prompt leverages definitions from Table 2 along with illustrative examples to guide few-shot prompting within the LLM pipeline. The prompt details the output formatting rules and provides examples to promote compliance.

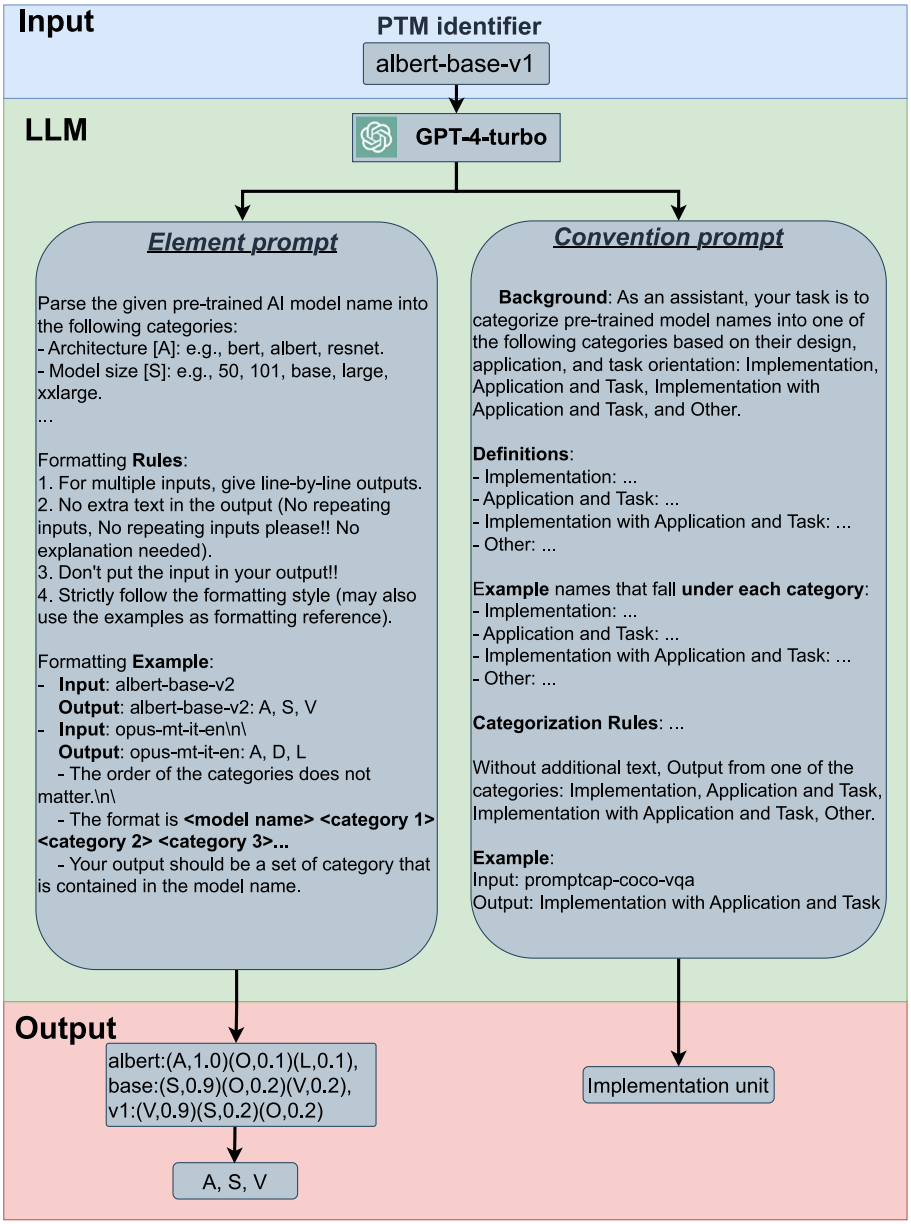

Fig. 4 Example of input/output of GPT-4. Input is a batch of PTM package names. The prompts include both the definition of naming elements (Table 2) and conventions (Table 9). In this example, A represents the **A**rchitecture (*e.g.,*Albert), S represents the model **S**ize (*e.g.,*base), and V represents the **V**ersion (*e.g.,*v1). Full versions of both prompts are available in Data Availability

– **Naming Convention Prompt:** The prompt begins by defining all four naming convention categories (Table 9) and specify the relevant naming elements under each convention (*e.g.,*architecture, model size, dataset, versioning) that contribute to each category. A curated list of examples further supports few-shot reasoning, and the prompt concludes with explicit output rules and an end-to-end example to ensure consistent formatting.

The final prompts are available in our artifact (Data Availability). We report our final evaluation results in §5.2.3.

5.2.3 Method of Evaluation

We evaluated the reliability of our mining approach by comparing the LLM-generated outputs to manually labeled data for both naming elements and naming conventions. Two manually labeled datasets were used to assess the accuracy of the LLM pipeline.

For *naming elements*, we randomly sampled 300 models from the full list, providing a 95% confidence level. One researcher labeled the naming components by referring to model cards and consulted a second researcher for ambiguous cases. This labeling process was straightforward due to clear patterns in the model names and documentation.

Given the context-sensitive nature of *naming conventions*, two annotators followed our predefined labeling criteria and independently annotated 200 PTM package names. For instance, the naming element "language" could be categorized as either an implementation unit or an application, depending on the model context. The initial agreement between the two researchers, measured by Cohen's Kappa, was 0.72 (substantial agreement). Most disagreements were from confusion over abbreviations or borderline cases. The researchers also agreed to classify unclear cases as "other" category. After resolving these issues through discussion, they refined their labels and reached a final agreement of 0.86 (almost perfect agreement).

6 Results and Analysis for RQs 1-3

Here we present the results for Research Questions 1 (§6.1), 2 (§6.2), and 3 (§6.3).

6.1 RQ1: How is PTM naming different from traditional software package naming?

Finding 1: PTM naming is different from traditional naming.

88% of respondents perceive PTM naming to be different from traditional package naming. The main difference is the greater semantic knowledge embedded in PTM names. The main explanations for this difference were: (1) differences in evolution practices; and (2) different information needed for reuse decisions. and

A total of 52 respondents (out of 108) replied to the relevant subset of questions (Topic 2 in Table 4). We focused on the common case (88%): responses describing differences. We analyzed responses in two ways: *how* the names differ, and *why* the names differ.

Table 6 Induced themes and definitions for *how* PTM names differ from traditional names

Theme	Definition	# Participants (%)
SEM-KNOW	Contain more semantic knowledge	30 / 34 (88%)
MORE-VARIATION	More variation in PTM naming practices	5 / 34 (15%)
DIFF-KINDS	Kinds: Archetypal vs. aesthetic names	4 / 34 (12%)

Based on code-able responses from 34 participants. The key difference in PTM names is their higher semantic content compared to traditional packages

Table 7 Induced themes and definitions for *why* PTM names differ from traditional package names, based on coded responses from 37 participants

Theme	Definition	# Participants (%)
FORK	Evolution practice ("forking")	18 / 37 (49%)
REUSE	Information needed for reuse	18 / 37 (49%)
COMMS	Conventional location of semantic information	6 / 37 (16%)
VERSION	Definition of versioning	4 / 37 (11%)
NO-STD	No standardization, "Wild West"	5 / 37 (14%)
OTHER	Low frequent categories	7 / 37 (19%)

Half of the respondents identified evolution practices and essential reuse information as the key factors driving these differences

Table 6 summarizes the codes and response frequency related to *how* names differ between PTMs and traditional software packages. The primary difference was in the greater amount of semantic knowledge in PTM names, as depicted in Table 2. One subject described his observation: "*I see models being a whole other thing, where description and characterisation of a model within its name is highly beneficial, as many strive to solve the same goal in different ways*." To elaborate, he added: "*General package managers serve packaged and branded services/utilities and features… Naming things is good for recognition (e.g., Whisper, Llama), but I feel these models serve their purpose because they have become 'base models'… so any fine-tunings should be suffixed*." Another representative answer was "*Pre-trained model names often include more specific information about the model's architecture, training data, and performance. This...detail is not typically included in traditional...package names*." Other distinctions were that subjects observed greater variation in PTM names ("*The versioning system for models seems less well-developed*"), and that subjects saw different kinds of names between PTMs and traditional packages: "*PyPI can be sensible names, based on archetypal action, or fun names, based on authors' aesthetics*".

Table 7 summarizes the codes and response frequency related to *why* names differ between PTMs and traditional packages. The two main explanations were about evolution practices and reuse practices. One subject described the influence of evolutionary practices on names as follows: "*[Traditional] packages are not usually as directly built on previous ones, with small extensions added like for PTM. Finetuning models generally means that the outputs are similar to the base, unlike with traditional packages.*" With respect to the information needed for reuse, a representative quote was: "*Traditional software packages are supposed to contain an entire library...[with] various functions...and configuration... PTMs are generally trained for a single objective...Therefore, [PTM] naming is critical for readability and avoiding mistakes.*".

6.2 RQ2: What elements to include in PTM identifiers?

This section first validates our mining method (§6.2.1), and then gives an integrated answer to RQ2 combining our survey data and mining data (§6.2.2).

Finding 2: Participants prefer naming by architecture/function.

Survey participants preferred naming pre-trained models based on architectural characteristics ("how it works") and intended functions ("what it does"). There was less interest in incorporating training details ("how it was made").

6.2.1 Validation of Mining Method

As described in §5.2.3, we manually labeled naming elements and convention data, using these annotations as ground truth. We then compared the pipeline outputs against the manually labeled dataset to validate our pipeline. Table 8 shows that the naming element extraction achieved an F1-score of 91%, while Table 9 shows an F1-score of 89% for naming convention classification. These results indicate that the pipeline produces reliable outputs. Therefore, we applied the pipeline at scale to systematically measure naming practices in real-world PTMs, and compare with our survey results, as detailed in §6.2.2.

6.2.2 Comparison of Mining and Survey Results

Figure 5 illustrates survey participants' preferences (blue bars) for including specific elements in the naming of pre-trained models (PTMs). The elements most favored are related to the model's architectural lineage: "Architecture" (69/108, 63.9%), "Model size" (62/108, 57.4%), and "Task" (57/108, 52.8%). This highlights a strong preference for names that convey the model's high-level implementation details, facilitating easy identification and distinction, This is usually a starting point of PTM reuse as indicated by Jiang et al. (2023b).

Table 8 Evaluation of naming element measurements

We evaluated the results from 300 randomly selected names, achieving a 95% confidence level with a 5% margin of error. The definition of naming elements are presented in Table 2. *NOTE: No existing tools or data are available, so our manual analysis serves as the baseline*

Element	Precision	Recall	F1-Score
Architecture [A]	0.98	0.98	0.98
Model size [S]	0.96	1.00	0.98
Dataset [D]	0.94	0.89	0.92
Dataset characteristics [C]	1.00	0.74	0.85
Model versioning [V]	1.00	0.94	0.97
Reuse method [F]	1.00	0.87	0.93
Language [L]	0.95	0.92	0.94
Task [T]	0.90	0.91	0.91
Training process [R]	0.80	0.80	0.80
Number of layers [Y]	0.67	1.00	0.80
Number of params. [P]	1.00	0.96	0.98
Other [O]	0.87	0.81	0.84
Average	0.92	0.90	0.91

Table 9 Evaluation of naming convention measurements

Naming Convention	Precision	Recall	F1-Score
Impl. unit	0.79	0.94	0.86
App. or Task	1.00	1.00	1.00
Impl. + App./Task	0.92	0.73	0.81
Other	1.00	1.00	1.00
Average	0.90	0.89	0.89

We evaluated the results of 200 randomly selected names, achieving a 95% confidence level with a 7% margin of error. The data were manually labeled, and the results were discussed by two researchers to ensure quality. *NOTE: No existing tools or data are available, so our manual analysis serves as the baseline. Data is available in our artifact (Data Availability)*

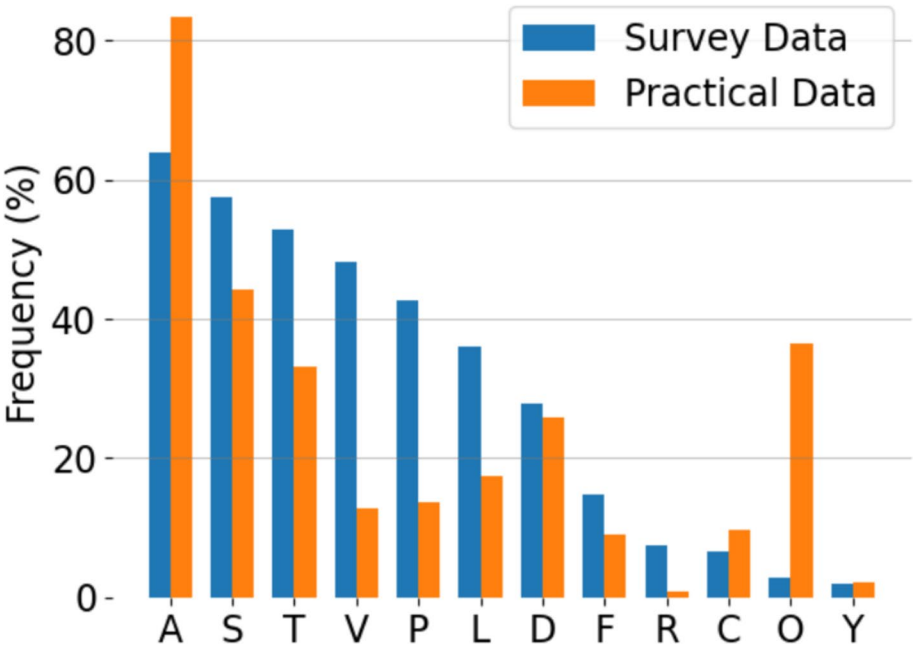

Fig. 5 The surveyed (blue) and measured (orange) distribution of naming elements. The x-axis refers to the codes from Table 2. The figure shows that naming practices do not consistently align with engineers' naming preferences

A smaller yet significant number of users want low-level details, including model versioning (52/108, 48.1%), number of parameters (46/108, 42.6%), language (39/108, 36.1%), and training dataset (30/108, 27.8%). These elements are relevant to model selection and evaluation.

Figure 5 also presents the practical distribution (orange bars) of naming elements. Comparing survey participant preferences (blue) with real PTM names (orange), we see that naming practices do not always match naming preferences. Almost all models (∼85%) include Architecture (A) information, though only ∼60% of respondents prefer it. As an example in the reverse direction, many respondents value information such as task (T), ver-

sion (V), and parameter count (P), but these elements are comparatively rare in real model names, appearing in only 32%, 11%, and 12% of names, respectively.

Finding 3: Design purpose is important in PTM naming.

The survey suggests a preference for names that combine details about both the implementation and the application or task (60.2%), while actual PTM packages tend to be named solely based on implementation units (59.0%).

Figure 6 displays engineers' preferences for naming conventions in the context of reusing PTMs, along with the measured practices using our LLM-based automated extraction pipeline. The most preferred convention (58, 60.2%) combines "implementation + application/task" (*e.g.,* `Llama-2-7b-chat-hf`), with "Application/task" following (41, 23.3%). This suggests engineers favor names that reflect both the intended purpose and the implementation specifics of PTM packages. These patterns are rare in practice (only ∼35% and ∼5% of models) — PTM names tend to share only implementation details (59%). PTM users' preferences are misaligned with naming practices.

6.3 RQ3: How do engineers identify inconsistent names?

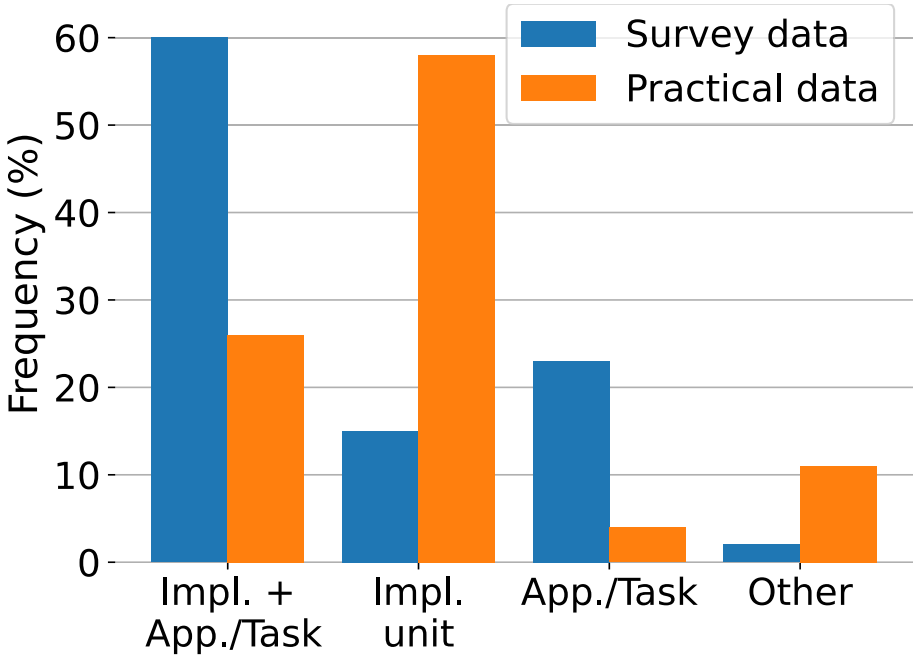

Fig. 6 The surveyed (blue) and measured (orange) distribution of naming conventions. Survey results indicate that design purpose is considered important in PTM naming, yet it is not widely reflected in practice

Finding 4: Users manually verify naming consistency.

PTM users identify naming inconsistencies by reading both internal (69%) and external (11%) information provided by the PTM package. Around half of users (46%) inspect it, *e.g.,* via measurement or visualization. They inspect both model metadata and architecture.

A total of 52 respondents replied to the survey question corresponding to this RQ (§5.1.4). Table 10 summarizes the codes and response frequency for how to identify PTM naming inconsistencies. PTM users described two kinds of approaches: (1) reading internal/external information about the PTM; and (2) downloading and inspecting the PTM.

Most participants mentioned that they check for inconsistencies by reading information provided by the PTM packages. A large proportion of the participants (36/52, 69%) mentioned that they identify the naming inconsistencies by *"reading the model cards"* and other metadata included with PTM, and comparing the data with the elements in PTM names. The metadata available in the `config.json` file is another primary source for verifying the accuracy of naming elements. That metadata is *"the ultimate source of truth because models can lack model cards"*. Some participants (11/52, 21%) also mentioned that they refer to external information about PTM, including research papers, discussion forums, and issue reports. The external information can provide the users with more detailed information, *e.g.,* whether other people have already verified the naming elements for that PTM.

Approximately half of the participants (24/52, 46%) mentioned that they inspect PTMs to identify naming inconsistencies. This inspection often involves *"visualizing/summarizing the model architecture"*, *"running [the model] on actual data/examples"*, and *"examining the memory usage"*. They noted that visualization or manual inspection makes it *"obvious if the architecture is different to what the model name says"*. When running inference, users can verify if *"the memory usage matches their expectations"* or *"if there is any substantially decreased performance"*. Detecting naming inconsistencies related to the task and application goals is also clear *"if the model is unable to perform the given task/achieve the mentioned goal"*. Although costly, these hands-on approaches ensure that the model's capabilities align with its description.

7 RQ4: How can architectural metadata inconsistencies in PTM naming be automatically detected?

Table 10 Actions for identifying PTM naming inconsistencies, mentioned by multiple participants

Action	# Participants (%)
Read metadata included with PTM	36 / 52 (69%)
Read external information about PTM	11 / 52 (21%)
Inspect the PTM	24 /52 (46%)

The third column shows the number of participants who mentioned each process. 52 (out of 108) participants answered this question. Users typically rely on manual metadata validation or model inspection to verify the correctness of a PTM name

Finding 5: Architectural information suffices to detect some kinds of naming inconsistency.

We developed an automated tool (DARA) to identify naming inconsistencies in PTM metadata based solely on architectural information. Our results show that the tool achieves high accuracy for `model_type` detection (99.0%), and performs moderately well for `architecture` (72.0%) and `task` (76.3%). These findings suggest that architectural features are reliable for broad model categorization but insufficient alone for more fine-grained classification tasks.

From RQ1 and RQ2, we found that PTM naming differs from traditional package naming, with a preference for names that reflect architectural characteristics (*i.e.,*architecture and model type) and intended functionality (*i.e.,*task). From RQ3, we learned that PTM users are concerned about naming inconsistencies, and that their current checking approaches are either cursory (reading metadata and external information) or costly (visual inspection or measurement of the PTM).

Here we investigate the automatic detection of naming inconsistencies. Early approaches to finding anomalous names in software were predominantly been rule-based (Høst and Østvold 2009; Pradel and Gross 2011; Liu et al. 2016). More recently, machine learning-based approaches have achieved state-of-the-art performance (Pradel and Sen 2018; Winkler et al. 2021; He et al. 2021a). For instance, Pradel et al. approached the problem of naming-based bug detection using binary classification (Pradel and Sen 2018). They proposed a method for identifying and correcting naming-related issues in source code by mining name patterns from extensive codebases. This approach uses a classifier trained with a small dataset to achieve high precision in real-world repositories (He et al. 2021a).

Inspired by these approaches, we propose a machine learning-based approach to detect anomalous names in PTM packages. We discuss feature selection (§7.1), feature extraction (§7.2), evaluation approach (§7.3), and results and analysis (§7.4).

7.1 Feature Selection

Our survey instrument included a set of questions on "*What kinds of modifications necessitate a new model type or architecture*" (Table 4). Table 11 presents a compilation of factors influencing the creation of new model types or architectures by PTM developers. More than one-third of the survey respondents believe that alterations such as modified training protocols, changes in input/output dimensions, layer additions or removals, modifications to the tokenizer, and shifts in the training dataset significantly contribute to the generation of new PTM names (Table 11). If the typical engineer acts on these beliefs, then their naming function could be mathematically represented as:

$$\mathcal{F}\left(\begin{array}{c}\text{layer connections}\\ \text{layer parameters}\\ \text{tokenizer}\\ \text{pre-trained weights}\\ \text{training regime}\\ \text{dataset}\end{array}\right) = \left(\begin{array}{c}\text{mode\_type}\\ \text{architecture}\\ \text{task}\end{array}\right) \tag{1}$$

Table 11 Factors influencing the need for new PTM names from reusers' perspectives

Over 30% of participants consider training regime, tensor shape, and layers in the main body as key factors for PTM name changes, while others hold no strong opinion

Theme	# Part. (%)
Modified training regime	43/108 (40%)
Modified tensor shape	34/108 (31%)
Modified layers in the main body	33/108 (31%)
Addition/deletion of layers in the main body	33/108 (31%)
Changed training dataset	31/108 (29%)
Modified tokenizer	31/108 (29%)
Addition/deletion of layers in IN/OUT layers	26/108 (24%)
Other	4/108 (4%)

The survey data suggests that the `model_type` and `architecture` are determined by several factors §6.3. However, prior studies on PTM reuse suggested that there is a lot of missing metadata from the model cards on Hugging Face (Jiang et al. 2023c, 2024). Hugging Face does not provide enough details about training regime and much dataset information is missing; the non-transparency reduced the trustworthiness and reusability of PTM packages (Jiang et al. 2023b). The primary information provided by Hugging Face encompasses tokenizer details, pre-trained weights, and the labels for `model_types` and `architecture`. Given this "data of convenience", we therefore formulate the following engineering hypothesis:

Engineering hypothesis: Using only architectural information, we can effectively detect architectural naming inconsistencies. In other words, we can learn the following function $\mathcal{F}$:

$$\mathcal{F}\begin{pmatrix}\text{layer connections}\\ \text{layer parameters}\end{pmatrix} \approx \begin{pmatrix}\text{model\_type}\\ \text{architecture}\\ \text{task}\end{pmatrix} \tag{2}$$

Following the guidance of Melegati et al. (2019) and Olsson and Bosch (2014), we frame this as an engineering hypothesis. Engineering hypotheses are formulated to guide the design and development of systems and can be evaluated through empirical testing (Melegati et al. 2019). They are design tools to be tested through system-building and evaluation, not traditional scientific hypotheses to be tested through statistical tests. Thus, to test this hypothesis, we designed and evaluated an automated technique that extracts layer connection and parameter information, and then learns the relation between these features and the human labels from Hugging Face.

We assess this hypothesis individually for each key component of a model's name – model type, architecture, and task. We describe our basic design of the automated technique in §7.2, and further feature extraction methods in §7.2.4. We consider a performance of over 75% (heuristic) as supportive of the hypothesis, indicating that further refinement could make the approach feasible for a model registry to use to automatically detect naming inconsistencies.

7.2 Design of DNN Architecture Assessment (DARA)

7.2.1 Overview

Figure 7 presents the DNN ARchitecture Assessment (DARA) workflow. The process begins with open-source pre-trained weights. First, we load each weight and feed dummy inputs to outline the graph architecture of each Pre-trained Model (PTM). Next, we transform the computational graph into an abstract architecture and implement n-gram feature extraction. Utilizing these features, we train a CNN classifier to identify naming inconsistencies.

We made the decision to develop DARA pipeline to detect naming anomalies for the following three reasons:

1. Prior work has shown that model metadata and identifiers on Hugging Face are often manually created and may be incomplete or inconsistent (Jiang et al. 2023b; Montes et al. 2022). Consequently, relying solely on such metadata is unreliable.
2. We view the pre-trained weights as the core components of PTM packages, similar to code modules in traditional software, since models function as executable code (Zhao et al. 2024; Ayodele 2010). Thus, the pre-trained weight file can be relied upon as the trustworthy element in a PTM package — not that it can be trusted, necessarily, but that it is the incontrovertible fact beneath the PTM's name.
3. There is no off-the-shelf approach that could be directly applied to pre-trained model packages for anomaly detection tasks.

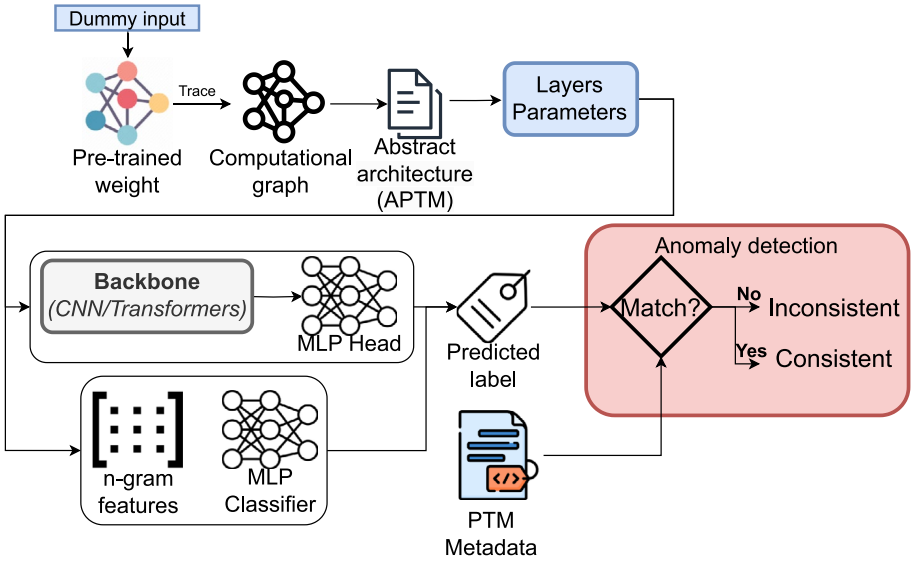

Fig. 7 To improve PTM naming accuracies, we propose the DNN ARchitecture Assessment pipeline (DARA), an anomaly detection algorithm designed to automatically identify discrepancies in PTM names. The approach involves detecting architectural variations across multiple PTMs, training a classifier, and using it to spot naming inconsistencies. Metadata inconsistencies are defined as mismatches between the actual architecture of a PTM and its corresponding metadata. In the DARA pipeline, these inconsistencies manifest as differences between the classifier's prediction and the labels provided by the PTM creator. We evaluate DARA using three key labels: `model_type`, `task`, and `architecture` (Fig. 2)

Therefore, we developed a pipeline from scratch that can take pre-trained weight files as input and identify naming inconsistencies with an anomaly detection approach (§7.2).

In this section, we present our DNN architecture assessment pipeline, including computational graph conversion, delineation of abstract architecture, multiple feature selection methods, and CNN classifier head for anomaly detection.

7.2.2 Computational Graph Conversion

We developed an automated pipeline to transform pre-trained weights into computational graphs. Existing work shows that PyTorch is the dominant framework in Hugging Face (Castaño et al. 2023), and using Hugging Face API we will load the PyTorch model by default (Hugging Face 2023). Recent studies confirm that PyTorch is the most commonly used framework on Hugging Face, followed by TensorFlow and JAX (Castaño et al. 2023). According to the PeaTMOSS dataset (Jiang et al. 2024), 96% of models use frameworks such as PyTorch, TensorFlow, or SafeTensor — all primarily designed for DNN development. This estimate is based on a representative sample with a 99% confidence level and a 5% margin of error. Our pipeline supports both dynamic graphs (*e.g.,*PyTorch), and static graphs (*e.g.,*TensorFlow, ONNX).

General Approach During graph conversion, our method is to visit each node in the model architecture and traverse the edges. The detailed algorithm is shown in Algorithm 1. When converting models with cycles like *Gated Recurrent Neural Networks* (Chung et al. 2014), we skip tracing upstream or backward edges after their initial visit. This approach turns every network into a *Directed Acyclic Graph* (DAG) with complete reachability, forming a **rooted DAG** in which each node lies on a path from an input to an output. This approach allows us to maintain the integrity and functionality of the graph while ensuring all nodes are reached efficiently.

For Dynamic Graphs (PyTorch) Due to PyTorch's dynamic structure, we introduce *dummy inputs* to the models to construct and trace the computational graphs effectively (Preferred Networks 2021). We refined our automated pipeline by iteratively adjusting *dummy inputs*: if a conversion failed due to the input, we incorporated a new input type into our input list. Consequently, our finalized pipeline employs fixed language inputs while utilizing random tensors for models in the vision and audio domains. The associated threat is discussed in §8.

For Static Graphs (*e.g.,*TensorFlow, ONNX) We load each framework's serialized model file and parse its graph definition directly via the native API (*e.g.,* `tf.GraphDef` for TensorFlow, `onnx.ModelProto` for ONNX). Since these formats explicitly enumerate nodes and their inputs, we extract the node list and adjacency information without runtime tracing. Control-flow constructs (loops, conditionals) are unfolded where supported, and unsupported ops are logged and omitted. This direct graph parsing yields a complete DAG representation ready for downstream analysis.

```
Input: Pre-trained weights of a DNN model
Output: List of layers sorted deterministically
Visited ←∅
model ←load_model(weights)
Function traverse(layer):
    // Recursively traverse the model graph to assign structural
       hash values to each layer
    if layer ∈ Visited then
        return
    Visited ← Visited ∪ {layer}
    if layer.connections is empty then
        layer.hash ← hash(layer)
    else
        foreach next_layer ∈ sort(layer.connections) do
            traverse(next_layer)
        layer.hash ← hash(layer, sum(next_layer.hashes))
foreach input_layer ∈ model do
    traverse(input_layer)
SortedLayers ← sort_by_hash(model.layers)
return SortedLayers
```

Algorithm 1 Deterministic layer sorting for model serialization in DARA. This algorithm computes hash values for each layer in a DNN based on its structure and dependencies, enabling a consistent serialization order. Determinism ensures reproducibility and meaningful comparison across models.

7.2.3 Abstract Architecture

To enable consistent model representation and facilitate downstream analysis such as feature extraction and reuse, we introduce the concept of an "*abstract architecture*". Prior work proposed *abstract neural network* (ANN) (Nguyen et al. 2022). We propose a similar concept, *abstract PTM architecture* (APTM). This new concept not only includes information about layers, parameters, and input/output dimensions (as ANN did), but also incorporates detailed information about the connections between nodes, offering a fuller depiction of the model's architecture. This enhancement is pivotal for our n-gram feature selection method. As part of our evaluation, we ablate to show the value of the APTM representation on this problem.

7.2.4 Feature Extraction Methods

To extract features from the abstract architecture (*i.e.,*the edge connectivity in model graphs), we explore two types of features extraction in this work: (1) basic *n-gram* feature extraction, and (2) advanced feature extraction utilizing pre-trained *transformers and contrastive learning*. Both approaches are typical representation learning approaches (Bengio et al. 2013).

We first experimented with using the raw architecture graph directly. This motivated our use of n-gram features, which efficiently capture frequently occurring local patterns in PTM architectures.

Next, we linearized the architecture graphs by applying traversal and topological sorting, producing consistent and semantically meaningful sequences of layer names. To evaluate how well these sequences preserve architectural semantics, we considered multiple classi-

fier architectures. Specifically, we experimented with CNNs and transformer-based models. These linearized sequences can be processed analogously to natural language, where CNNs are effective at capturing local architectural patterns (LeCun et al. 2015), while transformers model global dependencies across the sequence (Vaswani 2017).

While many powerful transformer variants exist, our goal in this study was to assess the effectiveness of different feature extraction paradigms – rather than exhaustively benchmark all possible model backbones.

(1) Basic Feature Extraction: N-gram We propose a method for vectorizing PTM architectural information using two fundamental components that define their architecture: layer connections, and layer parameters. These aspects include the structural configuration of each PTM, such as pairs of connected layers (*e.g.,* `(Conv2D, BatchNormalization)`), and specific layer attributes (*e.g.,* `<kernel_size: [1, 1] >` in a Conv2D layer). Following prior works (Ali et al. 2020; Terdchanakul et al. 2017; Sabor et al. 2017), we apply an N-gram feature extraction method to convert the architectural characteristics of each DNN into vector form. Specifically, we employ 2-grams for analyzing layer connections and 1-grams for detailing layer parameter information. Our goal is to find a good unsupervised feature for PTM architectures accurately and efficiently (Zhu et al. 2015). We also experimented with 3-gram methods for feature extraction; however, we found that the process was notably slow, and the resulting features were overly sparse and high-dimensional, suggesting this method is unsuitable for extracting features from PTM architectures. Subsequently, we apply padding to standardize the length of features across all PTMs. This innovative approach provides a structured and detailed basis for analyzing and comparing PTM architectures. An example of the extracted n-gram features is shown in Listing 1. These features are then converted into a one-hot encoded vector over all observed connection types. A visualization of the resulting feature representations across our sampled data is provided in Fig. 8.

We chose n-gram graph representation over Graph Neural Networks (GNNs) (Wu et al. 2020) for three primary reasons:

- *Simplicity*: Our n-gram graph method is equivalent to a simple graph neural network (Ma et al. 2022). Initially, we pursued this simpler approach to test our hypothesis, and it performed well enough that we decided to report it to the broader community. The n-gram approach yielded informative results for different task types, guiding future research directions. N-grams effectively represent sequential text data, which translates well to capturing the architectural information of PTM packages in our context. Our method mirrors approaches used in other domains, such as molecular data analysis (Liu et al. 2019a) and text classification over high-frequency data streams (Violos et al. 2018). In these instances, n-grams offer a compact representation of graphs equivalent to a basic GNN but without the necessity for extensive training. Additionally, as reported by Liu et al. (2019a), the representation construction time using the n-gram method is faster than that of GNNs, significantly reducing training times for our DARA implementation.
- *Computational complexity*: We considered using GNNs for our analysis. While GNNs can manage moderate graphs with reasonable node and edge counts, their computational efficiency decreases as the graph size increases, especially with models like LLMs that

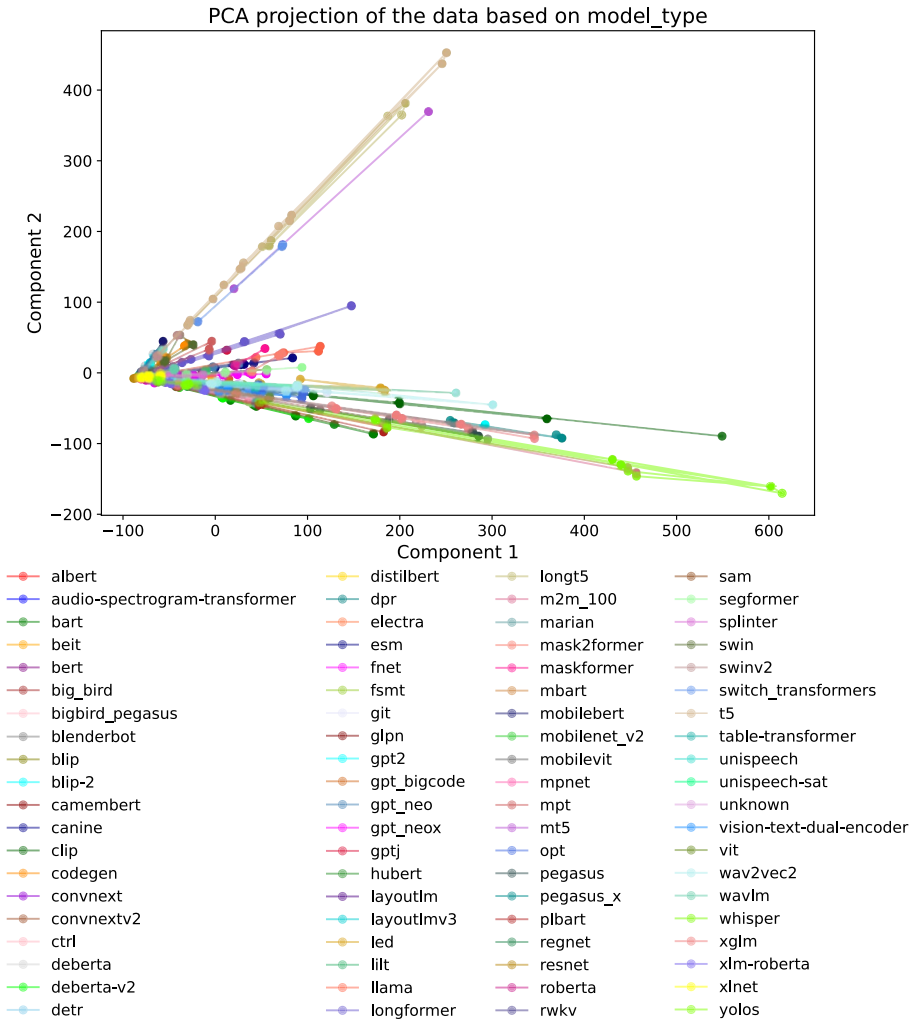

Fig. 8 PCA projection of our high-dimensional feature vector (4,304 dimensions in our dataset of 4108 PTM packages), visualizing `model_type`. The plot demonstrates that at least some `model_types` should be distinguishable using our feature extraction method

have thousands of nodes and edges (Ma et al. 2022). For instance, the `Mistral-7B-v0.3` model[8] has 1577 nodes and 1864 edges. This model, the smallest among the Mistral family at time of writing, illustrates the complexity that would burden GNNs at scale. Given the size of our dataset, we believe that the GNN method would introduce more computational overhead. Since one use case for DARA is as a check at PTM upload time, we think that registries such as Hugging Face would prefer using an automated tool for verification of the uploaded artifact. This is similar to the malicious package detection on NPM (Sejfia and Schäfer 2022) or automatically measuring performance goals in other registries such as the Android app store (Wu et al. 2017).

[8] See https://huggingface.co/mistralai/Mistral-7B-v0.3.

– *Feature dimension*: GNNs primarily extract features from the graph structure itself, making it difficult to incorporate specific layer parameters like kernel size in convolutional layers and dropout rate in dropout layers. In contrast, n-gram features can capture both layer connection information and detailed layer parameter information. These parameters are essential architectural details that we aim to consider in our analysis.

```
{
  "l": {
    "([INPUT], to)": 2,
    "(LayerNorm, Linear)": 259,
    "(Linear, ReLU)": 32,
    "(to, Conv2d)": 1,
    "(Conv2d, flatten)": 1,
    "...": "..."
  },
  "p": {
    "...": ...,
    "Linear ['<in_features, 4096>', '<out_features, 4096>']": 128,
    "LayerNorm ['<normalized_shape, (4096,)>', '<eps, 1e-05>', '<elementwise_affine, True>']": 65,
    "Conv2d ['<in_channels, 3>', '<out_channels, 1408>', '<kernel_size, (14, 14)>', '<stride, (14, 14)>', '<padding, (0, 0)>', '<dilation, (1, 1)>', '<transposed, False>', '<output_padding, (0, 0)>', '<groups, 1>', '<padding_mode, zeros>']": 1,
    "...": ...
  }
}
```

Listing 1 Example of extracted n-gram features from the Hugging Face model `Salesforce/blip2-opt-6.7b-coco`. The "l" field encodes the frequency of consecutive layer-type pairs (*e.g.,*(Linear, ReLU)), while the "p" field captures parameter-level patterns within individual layers. This representation is then converted into a one-hot vector and used as input for our downstream anomaly detection analysis.

(2) Advanced Feature Extraction: CNN, Transformers, and Contrastive Learning To achieve better representation of model architectures, we explored various feature extraction methods, including transformer-based backbones and contrastive learning approaches, to replace the N-gram approach. Our process of computational graph conversion and abstract architecture remain the same in this advanced approach.

Transformers excel at capturing complex dependencies in sequences using attention mechanisms, enabling them to model intricate relationships between model layers (Vaswani 2017). Contrastive learning, on the other hand, is effective for learning discriminative representations by maximizing the similarity between positive pairs and minimizing it between negative pairs (Jain and Verma 2021; Saieva et al. 2023; Huang and Lin 2023). In our approach, we represent model architectures as text sequences composed of their layer names. These sequences are then processed by transformers, while contrastive learning is applied to capture meaningful representations of the architectures.

Transformer-based Feature Extraction Since APTM is already sorted by Algorithm 1, we can directly extract a sequence of ordered layer names from each model, as illustrated in Listing 2. Transformer-based models are highly effective at modeling the complex, contextual relationships within sequences, making them well-suited for processing tokenized PTM layer names in such format. Following prior work (Huang and Lin 2023), we chose `RoBERTa` due to its efficiency and adaptability (Liu et al. 2019b). RoBERTa's deep contextual embeddings enable our model to capture subtle differences in PTM layer naming conventions, which is essential for downstream classification tasks.

However, RoBERTa has a maximum input token limit of 512, restricting the extracted features to only certain parts of the PTM package architecture. To overcome this, we experimented with the `Longformer` model, which supports a maximum token length of 4096 (Beltagy et al. 2020). By increasing the token limit, Longformer eliminates the need for input trimming, preserving more crucial layer information. This approach proved more effective than RoBERTa, as it enhanced performance by retaining essential details that would otherwise be lost due to token truncation.

```
{
  "Salesforce/blip2-opt-6.7b-coco": {
    "layers": "Input to Conv2d flatten transpose cat add
        LayerNorm Linear reshape permute __getitem__
       matmul mul softmax Dropout matmul permute reshape
        Linear add ... Output Input Embedding to",
    "model_type": "blip-2",
    "arch": "Blip2ForConditionalGeneration",
    "task": [
      "text2text-generation",
      "image-to-text",
      "visual-question-answering"
    ]
  }
}
```

Listing 2 Example of our advanced sequential features and associated architectural metadata extracted from the Hugging Face model `Salesforce/blip2-opt-6.7b-coco`. The listing shows a linearized sequence of layer names, which we use to represent architectural structure in a format suitable for downstream classification.

Contrastive Learning Our approach leverages Contrastive Learning (CL) to group together PTMs with the same labels and to separate those with different labels in the embedding space. To construct features for the CL pipeline, we extract and tokenized the names of individual layers from each PTM. Preprocessing begins by separating layer names with whitespace, followed by tokenization.

The core of our CL feature selection strategy is built on transformer-based encoders, augmented with task-specific linear classification head to predict model type, task, and architecture. We extract the final hidden representation of the special `[CLS]` token from each sequence and apply l_2 normalization to obtain the embedding used in the loss computation.

To leverage the architectural metadata in PTM packages, we adopt Supervised Contrastive Learning (SupCon) loss (Khosla et al. 2021), which extends traditional contrastive learning by leveraging label supervision to pull together examples from the same class

while pushing apart those from different classes. This approach allows each anchor to be contrasted against multiple positive samples, in addition to a diverse set of negatives, leading to more discriminative and robust representations.

$$\mathcal{L}_{\text{SupCon}} = \sum_{i=1}^{N} \frac{-1}{|P(i)|} \sum_{p \in P(i)} \log \left(\frac{\exp\left(z_i \cdot z_p / \tau\right)}{\sum_{a \in A(i)} \exp\left(z_i \cdot z_a / \tau\right)} \right) \tag{2}$$

Here, N is the batch size, with $\{x_i, y_i\}_{i=1}^{N}$ denoting input-label pairs. The embedding $z_i \in \mathbb{R}^d$ is the l_2 normalized final hidden representation of the special [CLS] token from the encoder. For each anchor z_i, $P_i \subseteq A(i)$ denotes the set of positive indices, where $P(i) = \{p \in A(i) : y_p = y_i\}$, representing the indices of all positive samples in the batch that share the same sample as y_i (excluding y_i itself). $A(i) = \{1, \ldots, N\} \setminus \{i\}$ is the set of all other samples in the batch. τ is the temperature, a hyperparameter used to scale the logits, and can be tuned to scale the penalties on negative samples.

For task metadata, some models can have multiple tasks. Therefore, we consider it as a multi-label classification problem for detecting task inconsistencies. We employ Multi-Label Supervised Contrastive (MultiSupCon) loss (Zaigrajew and Zieba 2022), which relaxes the constraint of positive instances by treating samples as positives if they share any overlapping label. The set of positive indices are defined as $P'(i) = \{p \in A(i) : s_{i,p} \geq c\}$, where $c \in [0, 1]$ is a hyperparameter representing the threshold value that indicates how similar sets of labels should be considered positive. MultiSupCon is defined as

$$\mathcal{L}_{\text{MultiSupCon}} = \sum_{i=1}^{N} \frac{-1}{|P'(i)|} \sum_{p \in P'(i)} s_{i,p} \cdot \log \left(\frac{\exp\left(z_i \cdot z_p / \tau\right)}{\sum_{a \in A(i)} \exp\left(z_i \cdot z_a / \tau\right)} \right) \tag{3}$$

where $s_{i,p}$ represents the weight in the assimilation value. This value indicates how similar two sets of ground truth labels are, with the value of 0 meaning zero overlap and 1 meaning perfect match.

To train the model, we adopt a joint training strategy that combines CL with cross-entropy (CE) loss. The CL component encourages semantically similar samples to cluster in the embedding space, while the CE component optimizes for label accuracy (Gunel et al. 2020): For model type and architecture metadata inconsistency detection, we use SupCon loss ($\mathcal{L}_{SupCon}$) and standard cross-entropy loss ($\mathcal{L}_{CE}$):

$$\mathcal{L} = \lambda \cdot \mathcal{L}_{SupCon} + (1 - \lambda) \cdot \mathcal{L}_{CE} \tag{4}$$

For task metadata inconsistency detection, we apply MultiSupCon loss ($\mathcal{L}_{MultiSupCon}$) and binary cross-entropy loss ($\mathcal{L}_{BCE}$):

$$\mathcal{L} = \lambda \cdot \mathcal{L}_{MultiSupCon} + (1 - \lambda) \cdot \mathcal{L}_{BCE} \tag{5}$$

7.3 Evaluation

In this section, we demonstrate the *effectiveness* of our proposed anomaly detection method (DARA). We note that this is the first known approach to detect PTM naming inconsistencies so there is no baseline for comparison. Our evaluation focuses on three key naming labels: `model_type`, `task` and `architecture` (§2.1).

7.3.1 Evaluation Dataset

We evaluate DARA on sampled data from the PeaTMOSS dataset by calculating the accuracy of the prediction of the trained classifier (Jiang et al. 2024). There are totally 219 architectures which have over 20 PTM instances. For those with more than 20 downloads, we randomly selected models to match 30 PTM instances. If fewer than 30 instances were available, we randomly sampled models with fewer than 20 downloads. During feature extraction, we were unable to load the PTMs from four architectures, *e.g.,* Bloom (Le Scao et al. 2022), because their model sizes are too large to be loaded. We were also unable to load another 2212 PTMs across these categories because either our dummy inputs could not be used on those models, missing configuration files or their architecture included customized code which we cannot fully trust.

For `model_types` and `architectures` labels, we utilized metadata available in the `config.json` file. For `task` label, we identify supported tasks for each PTM by parsing the task tags listed in the model card metadata. Any data entries without task tags were excluded from the analysis.

Overall, we collected 4108 PTM packages from 184 `architectures`, 81 `model_types`, and 22 `tasks`. We implemented an 80-20 split for training and evaluation purposes (Géron 2019). Each model in the dataset has its corresponding APTM and architectural metadata as labels (*i.e.,*model_type, architecture, task). To ensure robust validation, we employed 5-fold cross-validation on the shuffled dataset.

7.3.2 Evaluation Design

Table 12 summarizes the model variants evaluated in our study. When designing DARA's pipeline, we systematically evaluated the contributions of each key component, including feature extraction methods, model architectures, and training strategies. We started with a simple classifier using n-gram features, and then incrementally modified individual components to assess their impact. We summarized all ablation experiments below:

1. **N-gram + MLP Classifier:** Training a simple FCN classifier with N-gram features. The classifier consists of four fully connected layers. We systematically implemented a grid search to optimize the hyperparameters for DARA. The final training parameters include 50 epochs, a learning rate of $1e-3$, Cross Entropy loss, and a step-wise LR scheduler. Batch sizes are 256 for training and 32 for evaluation. "`l` " denotes layer connections; "`p` " denotes layer parameters. For our ablation study, we evaluated two configurations: `l+p` and `l` only.
2. **CNN:** Applying a Convolutional Neural Network to capture local patterns in the sequential model architecture. This setup treats the architecture as a 1D sequence and

Table 12 Summary of DARA model variants and the rationale for considering each

Model Variant	Design Rationale
N-gram + MLP Classifier (Wang and Manning 2012; Zhang et al. 2015)	Serves as a simple baseline using hand-crafted n-gram features (l, p) with a multi-layer perceptron. Fast, interpretable, and useful for benchmarking.
CNN (Kim 2014)	Applies 1D convolution over the architecture sequence to capture localized patterns in model structure. Effective for modeling short-range dependencies.
Transformers (Vanilla Finetuning) (Devlin et al. 2019; Liu et al. 2019b)	Fine-tunes pre-trained transformer models (*e.g.,*RoBERTa, Longformer) using cross-entropy loss. Provides strong general-purpose representations for classification tasks.
Transformers (Continued Pretraining) (Gururangan et al. 2020)	Continues masked language model pretraining on PTM sequences to better align transformer representations with domain-specific patterns.
Transformers (Finetuning w/ CL) (Khosla et al. 2021; Zaigrajew and Zieba 2022; Gunel et al. 2020)	Enhances feature discrimination by combining contrastive learning with cross-entropy loss. Improves representation quality through structured training objectives.

extracts spatially localized features using convolutional filters. Training hyper-parameters are identical to the N-gram approach.

3. **Transformers (Vanilla Finetuning):** Fine-tuning using cross-entropy loss alone, serving as a strong baseline for classification. The model is trained for a maximum of 50 epochs, a learning rate of 5e-5, Cross Entropy loss, step-wise LR scheduler, and early stopping with a patience of 5 epochs. For RoBERTa, batch sizes are 256 for training and 32 for evaluation. For Longformer, batch sizes are 8 for training.
4. **Transformers (Continued Pretraining):** Continued pre-training of transformer models on our data to adapt general representations to solve the model architecture problem. We continue pre-training using a masked language modeling objective. A pre-trained masked language model is further trained on our sequential PTM layer names using 15% token masking (Devlin et al. 2019; Liu et al. 2019b). The model is trained with vanilla fine-tuning with the same training hyper-parameters.
5. **Transformers (Finetuning w/ CL):** Fine-tuning using contrastive learning combined with cross-entropy loss to enhance representation quality. Training parameters are identical to vanilla finetuning setup. We performed a grid search over 8 combinations of the temperature parameter $\tau \in \{0.1, 0.3, 0.5, 0.7\}$ and $\lambda \in \{0.1, 0.3\}$. The best performance was observed with $\tau = 0.1$ and $\lambda = 0.1$, which we used for all `model_type`, `task`, `architecture` labels.

7.3.3 Anomaly Detection

As illustrated in Fig. 7, DARA applies the MLP classification head to prediction on the model_type, architecture, and task based on the features from the computational graph to make. detect anomalies in the PTM dataset by comparing predicted labels with the original PTM metadata. This strategy is motivated by its effectiveness in related tasks, such as time-series and log-based anomaly detection (Ren et al. 2019; Le and Zhang 2022).

7.4 Results and Analysis

7.4.1 Results

We conducted all our training and evaluation using a single `A100-80GB` GPU. Tables 13 and 14 present 5-fold cross-validation results for DARA on `model_type`, `architecture`, and `task` metadata prediction.

DARA achieves high accuracy in detecting inconsistencies in `model_type`, with top models (*i.e.,*Longformer with CL) reaching up to 99.0% accuracy. For `architecture`, DARA shows moderate performance, with the best transformer-based models achieving up to 72.0% accuracy, notably outperforming n-gram and CNN approaches.

In the task prediction task, DARA shows more modest accuracy. N-gram-based models outperform transformer-based ones on Top-1 predictions, achieving up to 56.5% F1 and 72.3% accuracy. This trend persists in Top-2 and Top-3 predictions, where N-gram models reach 61.0% F1 at 82.1% accuracy (Top-2) and 53.0% F1 at 90.0% accuracy (Top-3), outperforming transformer variants. These results suggest that shallow lexical features may better capture task-relevant patterns than deeper contextualized representations in this setting.

7.4.2 Analysis

Effectiveness of DARA Figure 9 shows the cases where DARA cannot predict correctly in `model_type`. In fact, these models share almost the same architectural designs and the only differences are either different training regimes, different datasets, or using different representations. This indicates that while architectural features are sufficient for model_type detection, they have limitations in distinguishing between architectures with similar structural patterns. To accurately differentiate between these architecturally similar but functionally distinct models, more sophisticated feature extraction approaches that capture training dynamics, data characteristics, and representational differences would be necessary.

Figure 10 shows the cases where DARA cannot predict correctly in `architectures`. The same incorrect case arises for the four Audio models. Furthermore, we can also find that DARA cannot correctly distinguish between different tasks as part of the `architecture`, such as `TokenClassification`, `QuestionAnswering`, and `SequenceClassification` in language models. Previous work indicates that the final layers of PTMs are highly task-specific (Rogers et al. 2021). Therefore, due to the high similarity of PTMs' features across these tasks, with differences confined to the final layers, DARA struggled to effectively differentiate between these categories. However, the Longformer, with its ability to handle longer input sequences and capture more global architectural features, offers improved performance in predicting the correct model architectures.

Figure 11 shows that DARA works well to predict the task label for computer vision tasks (*e.g.,* `image-classification`, `image-segmentation`, and `object-detection`), while struggles to accurately predict the tasks of PTM packages, particularly in distinguishing between `audio-classification` and `automatic-speech-recognition`. This difficulty likely stems from the similarity in their architectural configurations, where the main difference lies in the head or output layer of each model. However,

Table 13 5-fold cross-validation results for predicting model_type and architecture using DARA

DARA type	model_type				architecture			
	Recall/(%)	Prec./(%)	F1/(%)	Acc./(%)	Recall/(%)	Prec./(%)	F1/(%)	Acc./(%)
N-gram + MLP Classifier (l only)	96.2±1.1	96.4±1.1	96.1±1.0	95.7±0.4	60.9±0.8	56.1±2.2	55.4±1.4	62.2±0.7
N-gram + MLP Classifier (l+p)	97.6±1.2	97.6±1.3	97.5±1.2	98.8±0.3	62.8±1.2	59.8±1.6	58.2±1.3	63.7±1.3
CNN	88.5±1.5	89.1±2.0	88.0±1.8	92.1±1.1	65.1±2.6	64.9±2.4	62.1±2.4	66.2±1.4
RoBERTa (Vanilla Finetuning)	97.3±2.0	97.4±1.8	97.3±1.9	**99.0±0.3**	**70.4±1.4**	68.9±2.0	67.3±1.4	71.5±0.6
Longformer (Vanilla Finetuning)	97.8±2.1	97.8±2.0	97.7±2.1	**99.0±0.3**	70.3±2.3	69.2±3.3	67.4±2.6	71.6±1.1
RoBERTa (Continued Pretraining)	97.4±2.1	97.4±1.9	97.3±2.0	98.8±0.3	69.7±2.8	67.3±1.6	66.3±1.1	71.1±1.6
Longformer (Continued Pretraining)	97.8±2.1	97.9±2.1	97.8±2.2	99.0±0.5	70.2±1.6	68.8±1.7	67.4±1.5	**72.0±0.3**
RoBERTa (Finetuning w/ CL)	97.6±1.9	97.5±2.2	97.4±2.0	98.7±0.3	70.3±1.3	68.1±1.9	67.1±1.5	71.3±0.6
Longformer (Finetuning w/ CL)	**98.0±1.7**	**98.1±1.6**	**97.9±1.7**	**99.0±0.3**	70.2±1.2	**69.2±2.2**	**67.5±1.5**	71.9±1.3

Transformer-based approaches consistently outperform N-gram and CNN methods for both architectural metadata. Best performance is indicated in bold

these tasks might be more distinguishable by incorporating the weight information of the PTM packages, which is beyond the scope of this work.

The evaluation results of DARA demonstrate varying effectiveness across different metadata categories. For model_type classification, DARA achieves excellent performance with transformer-based approaches reaching up to 99.0% accuracy, strongly supporting our hypothesis that architectural information alone can reliably detect naming inconsistencies in this category. However, for architecture classification, the best performance reaches only 72.0% accuracy, and task classification performs even lower at approximately 76.3% accuracy for top@1 predictions. While we initially suggested 75% accuracy as sufficient for accepting the engineering hypothesis, our results indicate this threshold is met only for model_type detection. For architecture and task classifications, the lower accuracy suggests architectural information alone provides useful but not definitive signals for naming inconsistency detection. This reveals a gradient of detectability across metadata types, with model structure being most reliably identified from architectural patterns, while more specific attributes like task require additional contextual information beyond what DARA currently extracts.

Complexity Analysis We divide our time complexity analysis of DARA across five stages and provide both theoretical and empirical analysis. In our theoretical analysis, we use the following notation:

- n: Number of learnable parameters in a given PTM.
- V: Number of layers in the PTM graph.
- E: Number of edges, representing connections between layers in the PTM graph.

We empirically analyze each stage by fitting observed latencies against theoretical complexity curves, reporting the best-fitting model with the residual sum of squares (RSS). The results are presented in Fig. 12. All empirical measurements are conducted in terms of n, which correlates with and subsumes variations in V and E.

1. *Model Loading:* This stage is challenging to analyze, as it is influenced by both model architecture and Hugging Face's implementation of loading APIs. Empirically, the model-loading phase grows faster, fitting an $\mathrm{O}(n^k)$ curve most closely (RSS = 5.0091).
2. *APTM generation:* The APTM generation stage's complexity is based on the topological sort (Algorithm 1), which is $\mathrm{O}(V + E)$ with V the number of layers in a PTM graph and E the edges between layers (*i.e.,*layer connections). However, the linear regression analysis yielded an R^2 near zero, indicating negligible correlation between latency and model properties such as parameter count, layer count, or edge count. We conclude that in practice, latency is dominated by constant-time overheads, rather than any traversal or size-dependent computation. Empirically, the best match for this stage is $\mathrm{O}(\sqrt{n})$ (RSS = 10.022), slightly outperforming a linear fit of $\mathrm{O}(n)$ (RSS = 10.304), due to repeated dummy input calls and error handling.
3. *JSON export:* JSON export mostly depends on the total number of items of features. The theoretical time complexity for layer connections is $\mathrm{O}(E)$, layer parameters is $\mathrm{O}(V)$. Empirically, complexity follows $\mathrm{O}(n \log^2 n)$ (RSS = 1.2614), likely from bounded operation vocabularies and fixed vector structures.

Table 14 5-fold cross-validation results for predicting Task metadata using DARA

DARA type	task											
	Top@1				Top@2				Top@3			
	Recall/(%)	Prec./(%)	F1/(%)	Acc./(%)	Recall/(%)	Prec./(%)	F1/(%)	Acc./(%)	Recall/(%)	Prec.	F1	Acc./(%)
N-gram + MLP Classifier (l only)	**56.2±0.5**	**59.2±1.2**	**56.5**±0.7	72.3±1.2	**77.1±1.3**	**53.6±0.22**	**61.0±2.1**	82.1±0.8	**85.3±1.3**	42.9±1.2	53.0±1.5	90.0±0.9
N-gram + MLP Classifier (l+p)	55.6±0.5	57.7±1.3	55.9±0.8	71.8±0.5	76.0±0.8	51.0±1.6	58.8±1.0	82.9±0.4	85.7±1.9	40.5±2.5	51.3±1.9	90.5±1.0
CNN	52.6±2.6	58.1±3.5	53.8±2.6	73.5±1.3	75.1±4.1	45.5±3.1	54.0±3.1	82.1±1.7	81.4±3.3	34.3±2.7	44.5±2.3	88.3±1.0
RoBERTa (Vanilla Finetuning)	51.8±0.8	52.6±1.7	51.9±0.9	74.5±1.3	74.4±2.5	51.9±2.9	59.2±2.6	83.7±1.0	82.1±1.9	43.2±3.1	53.0±2.6	90.5±1.0
Longformer (Vanilla Finetuning)	54.4±2.7	58.0±4.2	55.1±2.8	**76.3**±1.4	75.1±1.8	52.2±1.4	59.4±0.9	**84.5**±0.6	83.5±2.4	43.3±2.9	53.6±1.9	90.6±0.8
RoBERTa (Continued Pretraining)	52.0±3.3	53.2±3.5	52.2±3.5	75.2±1.2	74.7±2.5	51.9±3.0	58.9±2.7	84.2±0.5	82.9±1.5	**44.0±3.1**	**53.7**±2.4	**90.8**±0.7
Longformer (Continued Pretraining)	53.4±2.5	57.0±2.7	53.8±2.3	75.0±4.5	75.5±3.5	51.1±1.9	58.4±2.3	83.0±4.2	82.3±4.5	42.4±2.4	52.4±2.5	89.6±3.0
RoBERTa (Finetuning w/ CL)	50.4±2.9	51.0±3.3	50.3±3.3	75.7±2.6	69.9±4.7	47.0±1.0	52.7±2.5	84.2±1.6	79.9±3.6	39.0±1.4	48.4±1.9	90.6±1.4
Longformer (Finetuning w/ CL)	45.8±5.7	46.8±5.9	45.4±6.2	68.9±7.3	65.3±8.8	39.9±5.9	46.8±6.7	80.2±5.4	75.9±7.9	33.7±3.3	42.7±5.0	88.0±5.1

N-gram-based approaches outperform transformer-based methods for task classification. Best performance is indicated in bold

4. *Feature Processing:* Converting features to IDs using a lookup dictionary has a time complexity of $O(V)$. Empirically, a linear fit $O(n)$ (RSS = 13.1625) matches observed latency.
5. *Prediction:* Once features are extracted, prediction is a constant-time lookup and classification step, $O(1)$. Empirically, this holds across all model variants, yielding minimal latency suitable for real-time use during PTM upload (Table 15).

When considering all stages from model loading through export—which comprise the majority of the entire pipeline—the cumulative end-to-end complexity again aligns with $O(n^k)$ (RSS = 4.9973).

Efficiency of DARA Figure 12 and Table 15 present the latency and throughput measurements for different DARA variants. The N-gram + MLP classifiers exhibit the lowest latency at 0.19 ms per PTM and the highest throughput, exceeding 28,000 PTMs per second, making them highly suitable for real-time deployment. The CNN model also performs efficiently, processing over 17,000 PTMs per second with a latency of 0.56 ms. Transformer-based models offer improved accuracy, particularly for `model_type` and `architecture` classification, but incur significantly higher computational costs. RoBERTa variants show moderate latency (8.4 ms) and throughput (740–750 PTMs/s), while Longformer variants are substantially slower, with latency exceeding 51 ms and throughput dropping to as low as 61 PTMs/s. These results confirm the computational feasibility of integrating DARA into the Hugging Face model upload workflow. Users would experience negligible delay when using N-gram or CNN-based variants, and acceptable latency with RoBERTa. The clear trade-off between accuracy and efficiency supports a tiered deployment strategy: fast models for broad coverage and real-time use, with more accurate transformer-based models reserved for high-stakes or ambiguous cases.

Validating Engineering Hypothesis The evaluation results reveal important limitations in our initial hypothesis about architectural features being sufficient for metadata validation. While we initially proposed a 75% accuracy threshold as acceptable for our engineering hypothesis, our results show a clear gradient of detectability across different metadata types. For model_type detection, with accuracy reaching 99.0%, the hypothesis is strongly supported. However, for architecture (72.0%) and task classification (76.3% top@1), the performance falls short of consistently meeting our target threshold. This indicates that architectural features alone provide valuable but incomplete signals for these more specific metadata categories. Rather than a binary acceptance or rejection of our hypothesis, these results suggest a more nuanced understanding: architectural patterns are highly reliable for broad model categorization but require supplementary information for more fine-grained classification tasks. This limitation highlights opportunities for future work in multimodal feature extraction that incorporates training dynamics and representational differences.

Generalizability of DARA To analyze the generalizability of DARA to recent PTMs, we conducted a stratified random sampling across the Hugging Face repository. We identified all publicly available models with at least 100 downloads and categorized them into five domains: NLP, CV, audio, multimodal, and other. Using proportional allocation based on domain distribution, we extracted a stratified sample of 1,901 models from the full reposi-

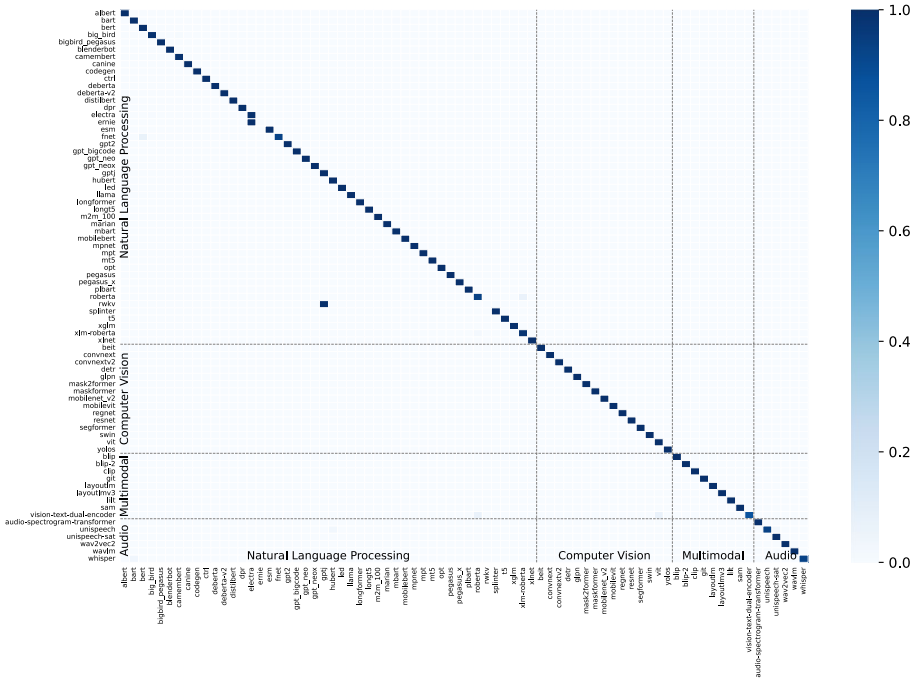

Fig. 9 Confusion matrix categorized by **model_type**. As seen along the diagonal of the confusion matrix, DARA accurately predicts most model types but encounters difficulties with audio models like *unispeech* and *wav2vec2*, which share the same architectural implementation and differ primarily in tokenization. A higher resolution version of this figure is available in the supplementary materials

tory (N=1,609,131), achieving a 99% confidence level with 3% margin of error. While our original dataset (§7.3) contained models ranging from 12K to 16B parameters, this latest sampling expanded to models from 26K to 70B parameters. For each sampled model, we simulated loading for APTM generation. As shown in Fig. 12, the red curves represent fitted distributions for each plot. The close alignment between these newer models and our fitted distributions demonstrates that our performance results generalize effectively to more recent models.

8 Threats to Validity

This section discusses three types of threats to validity (Wohlin et al. 2012). Following the guidance of Verdecchia et al. (2023), we emphasize substantive threats that could affect our findings.

Construct threats refer to potential limitations in how we operationalized the key concepts. *In the survey*, we operationalize the concept of PTM names in terms of the identifier and architecture. Alternative operationalizations might incorporate package type or package contents. However, per the registry naming taxonomy (Fig. 2), we believe our operationalization covers the elements that involve judgment on the part of the namer.

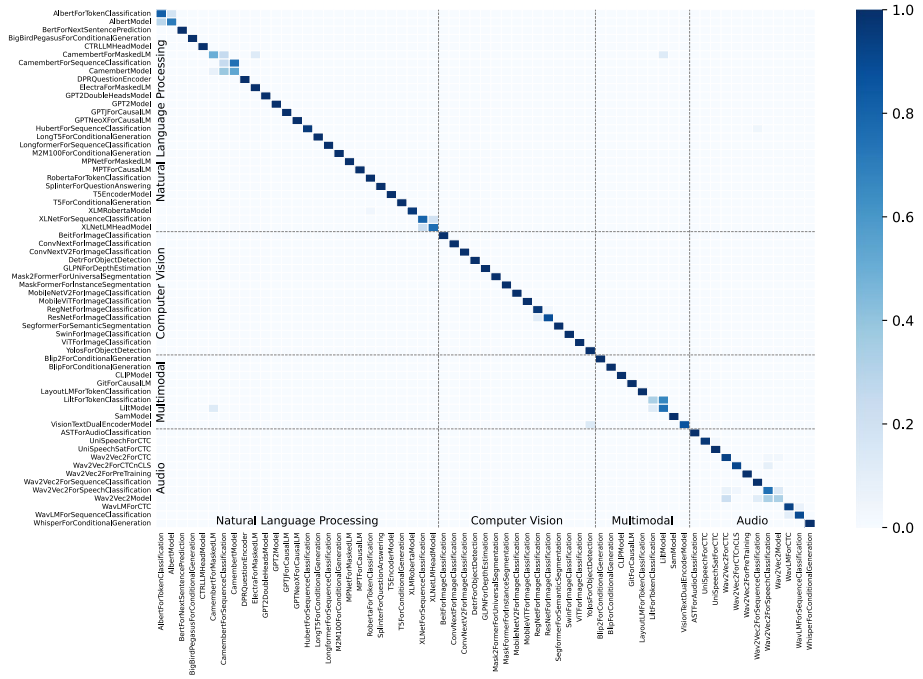

Fig. 10 Confusion matrix categorized by **architecture**. As seen along the diagonal of the confusion matrix, DARA accurately predicts most architectures, particularly for computer vision models, but faces challenges with similar tasks such as *QuestionAnswering* and *SequenceClassification*, where the primary difference lies in the model heads. A higher resolution version of this figure is available in the supplementary materials

In the other parts, the feature selection algorithm in DARA is designed based on an engineering hypothesis (§7.1). We validated this hypothesis by achieving substantial performance, indicating that the underlying assumptions were correct.

Internal threats are factors that may compromise the validity of cause-and-effect relationships. *In the survey*, a key threat is potential bias in the qualitative analysis of the two open-ended text questions. To mitigate this, two researchers independently analyzed the data, and resolved discrepancies through discussion, ensuring consistency in interpretation (§5.1.4). Another potential threat is the validity of our survey instrument. To address this, we conducted a pilot study with three actual Hugging Face users (§5.1.2).

In the other parts, we asked two researchers to manually label 200 naming conventions as ground truth in our measurement of naming conventions (§5.2.3 to evaluate our LLM pipeline. This process may introduce potential bias, but we measured inter-rater agreement, which indicated substantial consistency between the two researchers, mitigating this threat to some extent. Although DARA effectively detects metadata inconsistencies (§7.4), the algorithm's reliance on N-grams as features is a limitation, as it discards significant information about the PTM architecture and weights, reducing the depth of the analysis.

External threats are factors that may limit the generalizability of the findings. *In the survey*, our sample size of 108 achieves a 95% confidence level with a margin of error of 10%, based on the estimated number of Hugging Face users (§5.1.3). However, we recognize that

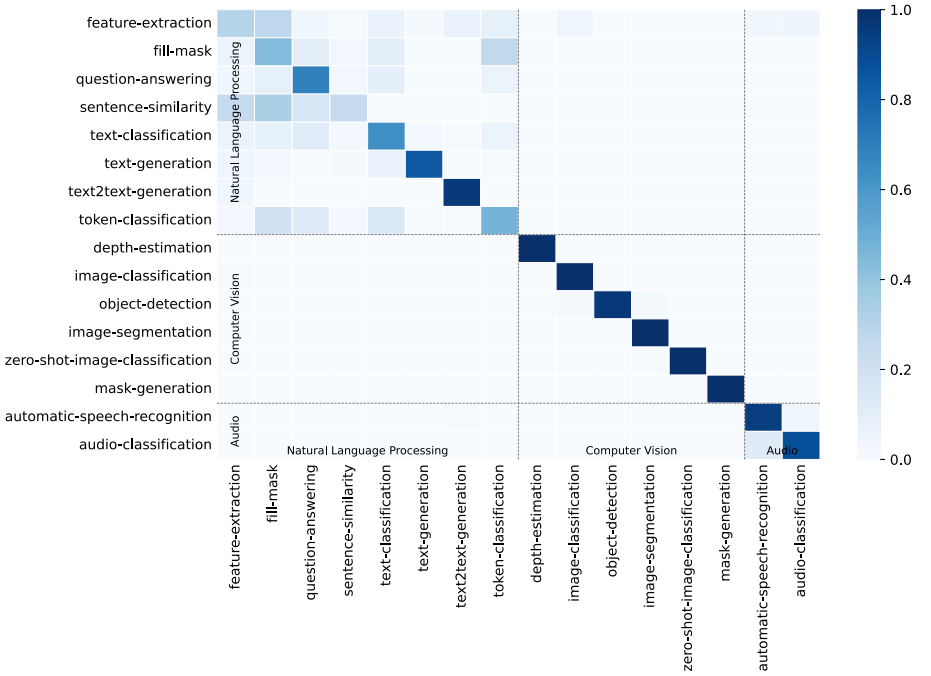

Fig. 11 Confusion matrix categorized by **task**. As seen along the diagonal of the confusion matrix, DARA accurately predicts vision tasks (*e.g.,*depth estimation and image classification), but performs less effectively for audio and text models. For clarity and visualization purposes, this confusion matrix only includes models with a single designated task (*i.e.,*excluding multi-task models)

using Hugging Face does not require a user account, meaning there may be significantly more users. Furthermore, since most participants were experienced PTM contributors or users, our findings may reflect a degree of selection bias. Additionally, our mining study focused on commonly used PTM packages with over 50 monthly downloads, and the exclusion of less popular packages may also introduce bias. These threats could potentially affect the generalizability of our study's findings. However, the particular bias that results — towards experienced engineers and towards packages that people actually use — strike us as desirable for a first study of PTM names. Future studies might investigate whether naming convention preferences and practices vary between experts and novices.

In the other parts, the dataset used for DARA evaluation is imbalanced, with a higher proportion of language and vision models and fewer multimodal and audio models. We recognize that this limitation may affect the generalizability of our evaluation. Additionally, we focused on attributes collected from PeaTMOSS, focusing primarily on data from the largest PTM ecosystem, Hugging Face (§5.2.3, §7.2). This introduces a slight risk that our findings may not generalize to other PTM ecosystems like ONNX Model Zoo or PyTorch Hub, and the August 2023 collection date might not capture the most recent PTMs. However, we expect that PTM practices remain relatively stable over a two-year period.

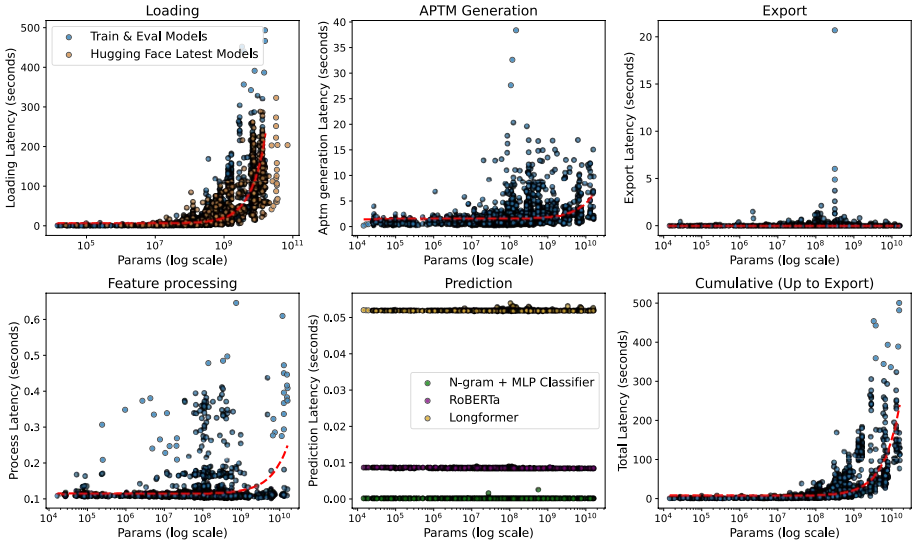

Fig. 12 Latency measurements for each stage of DARA, including APTM extraction, feature export, feature processing, DARA prediction, model loading, and total execution time. Red curves show fitted complexity trends based on our algorithmic analysis: quadratic for APTM extraction and feature processing; linear for feature export and prediction; and exponential for model loading. Orange markers indicate loading latencies of recent PTM/LLM models. The close alignment of these points with our fitted distributions demonstrates that our results generalize to newer models

Table 15 Comparison of DARA types by latency and throughput

DARA Type	Variant / Training Setup	Latency	Throughput	Macro F1 Score		
		(ms/PTM)	(PTM/s)	model_type	task	architecture
N-gram + MLP Classifier	layer connections	0.19	29849.12	96.1±1.0	**56.5±0.7**	55.4±1.4
	layer connections + layer parameters	0.19	28426.36	97.5±1.2	55.9±0.8	58.2±1.3
CNN	—	0.56	17127.99	88.0±1.8	53.8±2.6	62.1±2.4
RoBERTa	Vanilla Finetuning	8.43	750.91	97.3±1.9	51.9±0.9	67.3±1.4
	Continued Pretraining	8.34	740.11	97.3±2.0	52.2±3.5	66.3±1.1
	Finetuning w/ CL	8.43	748.18	97.4±2.0	50.3±3.3	67.1±1.5
Longformer	Vanilla Finetuning	51.60	64.96	97.7±2.1	55.1±2.8	67.4±2.6
	Continued Pretraining	51.78	61.58	97.8±2.2	53.8±2.3	67.4±1.5
	Finetuning w/ CL	51.90	64.58	**97.9±1.7**	45.4±6.2	**67.5±1.5**

Latency was measured by averaging inference time across individual models. Throughput was measured independently using batched inputs to reflect real-world usage. The final column shows the overall F1 score (*cf.* Tables 13, 14) to allow comparison of the efficiency/accuracy tradeoff. N-gram approaches yield the lowest latency, while transformer-based methods incur higher latency due to greater model complexity but offer better F1 performance

9 Discussion and Future Work

In this section, we describe the differences between PTM naming to traditional software naming (§9.1), the practical use cases of DARA (§9.2), and implication for PTM contributors (§9.3). We also outline three directions for future research: developing automated tools for package naming standardization, improving model selection and reuse, and strengthening PTM supply chain security (§9.4).

9.1 Comparing to Traditional Software

Prior research has demonstrated that inconsistent naming of components and software packages has hampered software reuse for decades (Krueger 1992; Griss 1993). Establishing more consistent and standardized naming conventions can improve software reuse (Hofmeister et al. 2017a; Alpern et al. 2024; Alsuhaibani et al. 2021). Our study offers a deeper understanding of naming practices in PTM packages and introduces an automated validation tool to detect naming inconsistencies. However, much work remains to be done.

What's in a name? We suggest that examining naming conventions has given us insight into the hybrid nature of PTMs as both reusable software artifacts and cataloged components with embedded metadata. Unlike traditional software packages that prioritize simplicity or branding, PTM names often encode critical architectural and functional details — more like a hardware spec sheet than a generic software identifier. This hybrid character underscores that PTMs are neither purely software nor hardware, but a unique class of reusable components whose names directly shape searchability, reliability, and reusability. As such, PTM naming is not just identifiers or metadata — it is interface.

9.2 Use Cases of DARA and Similar Automated Naming Tools

We describe DARA primarily as an **assessment tool** designed to detect naming inconsistencies, rather than a mechanism to enforce name changes. When DARA reports an anomaly, it signals a potential inconsistency between the actual model architecture and the metadata. This discrepancy warrants caution from users or platform operators. DARA generates labels for `model_type`, `task`, and `architecture` metadata, with mismatches indicating potential issues in PTM packages.

Some versions of DARA are lightweight enough to integrate into the Hugging Face model upload workflow. During model upload, an N-gram or CNN-based DARA could perform real-time consistency checks between the architectural features and the metadata provided by developers; they achieve throughput of 29,000 PTMs/sec and 17,000 PTMs/sec respectively (Fig. 12). This integration could follow a tiered approach:

- **Pre-upload verification**: As developers complete the metadata form for their model, DARA can analyze the model architecture in the background and suggest appropriate values for model_type, architecture, and task fields, potentially reducing manual errors.
- **Warning system**: When inconsistencies are detected, the platform can display non-blocking warnings to uploaders, allowing them to either confirm their unusual configuration or correct potential errors before publication.

– **Post-upload quality flagging**: For models already in the repository, periodic batch processing with more accurate (though computationally intensive) transformer-based DARA variants could identify and flag potential inconsistencies for platform moderators to review.

This deployment strategy balances efficiency and accuracy while keeping false positives to a minimum, as evidenced by our F1 scores for model_type (97.9%) and architecture (67.5%). The integration requires minimal effort from both platform operators (a single API endpoint) and model uploaders (who benefit from automated suggestions rather than facing additional requirements). The learning curve for using DARA is negligible, as it operates largely behind the scenes in the model upload workflow, with optional user-facing suggestions that follow familiar HF interface patterns.

DARA can be employed in various ways to enhance the efficiency and security of model registries and address problems that were discussed in §3. Below are three specific use cases that illustrate its potential applications:

– **Model Validation**: Platforms like Hugging Face could utilize DARA to validate models, ensuring that the names and metadata align with the actual architectures. This would enhance the transparency and trustworthiness of model registries by detecting discrepancies (Montes et al. 2022) and potential architectural backdoor attacks (Langford et al. 2024). Supporting a more secure AI model supply chain is crucial, as highlighted in recent technical reports (Hepworth et al. 2024; HiddenLayer 2024).
– **Label Generation and Validation**: DARA directly addresses the insufficient model card information and metadata issues (Jiang et al. 2023b; Taraghi et al. 2024) by analyzing architectural features during model registration. When a developer uploads a new model, DARA's classification pipeline extracts architectural patterns from model definition files and compares them against its trained classifiers to generate probable metadata values (model_type, architecture, task). These generated labels either validate developer-provided metadata or suggest appropriate alternatives when missing, improving model discoverability and reusability (Di Sipio et al. 2024; Jiang et al. 2023b). The process is fully automated, requiring no manual intervention while significantly enhancing metadata quality across the registry.
– **Plagiarism Detection**: When a new model is published, DARA can be used to detect similarities between the new model architecture and existing architectures. This functionality helps in identifying model architecture plagiarism, a topic of recent discussions (OpenBMB 2023).

Applying DARA in these scenarios can enhance the transparency, security, and efficiency of managing and validating AI models in model registries. However, DARA is not presented as a complete solution but rather as a proof of concept that highlights opportunities in this emerging problem domain. Our work tested a key engineering hypothesis, demonstrating strong results in specific areas while acknowledging certain limitations. DARA's implementation provides a foundation for further research and development, as discussed in §9.

9.3 Implications for PTM Contributors

Our study offers actionable guidance for PTM contributors, by identifying the major ways in which naming practices impact model reuse, selection, and ecosystem trust.

First, as detailed in §6.1, contributors should enrich PTM names with semantic details – such as architectural characteristics, parameter counts, and explicit reuse-relevant information – to better guide users. Given that 88% of survey participants noted that PTM naming conveys richer semantic content compared to traditional packages, if model authors ("name producers") embrace this more descriptive naming style then it will simplify model selection.

Second, the analysis in §6.2 shows a clear misalignment between user naming preferences and current naming practices. Users prefer names that reflect both the implementation ("how it works") and the application domain ("what it does"), yet many PTM names emphasize only technical specifications. Contributors can improve this disconnect by standardizing names to incorporate both dimensions. Using automated validation tools, as demonstrated by our DARA pipeline in §7.4 and discussed in §9.2, can help ensure that submitted names are consistent with the underlying model characteristics. In this way, automated linting integrated into PTM registries would not only facilitate a more uniform naming convention but also reduce downstream validation effort.

Third, our findings from §6.3 indicate that users currently rely on manual verification (*e.g.,*checking metadata and visual inspection) to detect naming inconsistencies — a process that is both error-prone and time-consuming. By embracing standardized naming conventions informed by our empirical analysis and by adopting automated checks during the model upload process, PTM contributors can minimize these inconsistencies. Improved consistency not only aids in model selection and reuse but also contributes to stronger supply chain security. In practice, clear and standardized naming could serve as an early warning mechanism against potential typosquatting or package confusion attacks and other malicious practices in the model ecosystem (Jiang et al. 2025; Ohm et al. 2020).

9.4 Future Work

9.4.1 Improving Naming Standardization through Automation

Poor naming practices lead to poor comprehension (Gresta et al. 2023; Alpern et al. 2024; Alsuhaibani et al. 2021). Our empirical analysis reveals that PTM naming differs from traditional package naming due to the substantial amount of semantic information embedded in PTM names (§6.1). However, the quality and consistency of PTM naming is insufficient, primarily due to a lack of standardized practices.

Automated validation tools to detect inconsistencies in uploaded packages would streamline the PTM reuse process. These tools would enhance the reliability of open-source packages while reducing the manual effort required for package validation by downstream users. Our DARA system is the first such tool. Although DARA effectively identifies real naming inconsistencies, its current feature extraction methods are limited (Tables 13, 14). For instance, DARA struggles to capture differences in training regimes and datasets between PTMs. One approach to address this might be stronger feature extraction techniques that convert model architectures into embeddings. Contrastive learning (Ni et al. 2021) offers

a potential way to address some of these limitations. Although it was not particularly successful in this context, we acknowledge that our application was exploratory. Contrastive learning might be more effective when combined with more advanced feature extraction (§7.2.4). We advocate for a more systematic investigation into the use of contrastive learning and advanced techniques, such as encoding directed graphs with transformers (Geisler et al. 2023), to enhance DARA's inconsistency detection capabilities by generating more robust feature representations.

In parallel, automated linting tools for package names could be introduced for PTM packages. While our work provides insights into PTM package naming, it does not yet address the best practices for standardization, akin to coding standards like Google's style guide (Google 2024b), PEP8 (Guido van Rossum et al. 2001), or C++ coding standards (Sutter and Alexandrescu 2004). As shown by Boogerd and Moonen (2008), the application of coding standards can enhance software reliability and reduce development effort, suggesting a similar benefit for naming conventions in package ecosystems. Considerable work has been done on developing linters to support software development, often integrated into continuous integration pipelines to automate tasks (Vassallo et al. 2020; Tómasdóttir et al. 2018; Habchi et al. 2018). In a similar fashion, linters for package names could be integrated into package contribution workflows (Sohacheski et al. 2021; Jiang et al. 2022a), improving standardization and enhancing both searchability and reliability. This approach seems particularly feasible for PTM registries, as our survey data suggests that contributors already perceive PTM names as being more standardized compared to traditional software packages (§6.1, §6.2). This existing trend toward standardization in PTM naming indicates that introducing automated tools like linters would be well-aligned with current practices, further promoting consistency and aiding discoverability.

Additionally, automated naming tools can support the development of PTM packages, much like existing techniques for generating class and method names in traditional software (Gao et al. 2019; Allamanis et al. 2015). Since PTM names often encode richer semantic information than conventional software names (§6.1), automatically generating names based on static analysis of modeling and configuration code could help developers produce meaningful, standardized names. This would not only reduce engineering overhead but also improve the consistency and clarity of PTM package naming.

9.4.2 Improving Package Naming for Better Model Selection and Reuse

The misalignment between engineers' expectations and PTM package naming conventions (§6.2), coupled with inconsistent package metadata, increases the engineering effort and costs associated with model selection and reuse (§6.3).

One approach to addressing the challenges of PTM selection and reuse is through automated machine learning (AutoML). AutoML techniques focus on automating aspects of model discovery and optimization, which can streamline the process and reduce manual effort (He et al. 2021b; Wang et al. 2023a). We propose leveraging tools like DARA, which has already demonstrated its ability to utilize model architecture meta-features for detecting metadata inconsistencies. By extending DARA to extract additional meta-features, such as intermediate outputs, we can enable large-scale AutoML processes that enhance model selection efficiency and simplify PTM adoption. For instance, adapting the model selection approach developed by Zhang et al. (2024) for specific downstream tasks allows us

to incorporate these meta-features, improving the selection process by considering engineering requirements like hardware constraints and domain adaptability (Patil et al. 2025). Nguyen et al. further demonstrated the value of meta-features from PTM implementations on GitHub in boosting AutoML performance (Nguyen et al. 2022). Other studies have also explored innovative model selection methods to support PTM reuse and improve AutoML's ability to identify optimal models (You et al. 2021a, b; Lu et al. 2019).

9.4.3 Securing Software Supply Chain by Leveraging Naming Information

Poor comprehension opens the door to mistakes. One class of attack related to naming is typosquatting attacks, which are known to occur in traditional software package registries (Gu et al. 2023; Neupane et al. 2023). For instance, attackers have exploited typographical errors in NPM package names to distribute malicious software, with several documented cases of typosquatting attacks targeting popular packages (ReversingLabs Threat Research Team 2023; Infosecurity Magazine 2023; Checkmarx Research Team 2023). Although we cannot point to a typosquatting incident for PTMs, *e.g.,*in Hugging Face, we suggest that we can learn from issues in traditional software to predict — and perhaps prevent — similar recurrence in this new domain. Solutions to hypothetical problems are not always necessary, but solutions to *predictable* problems strike us as a reasonable subject for research.

Recent studies have underscored other vulnerabilities in PTM packages, including backdoors and malware injection (Langford et al. 2024; Gu et al. 2019; Wang et al. 2022). These attacks threaten the AI model supply chain (HiddenLayer 2024). For example, backdoors can be introduced into PTMs by exploiting architectural variations, making them challenging to detect (Langford et al. 2024). Microsoft has emphasized that consistent naming practices can help expedite the detection and removal of such malware (Azure and contributors 2023). For instance, standardized naming conventions could enable PTM registries to develop automated tools that verify whether the package name accurately reflects the model's architecture, helping to identify potential backdoors. These tools could build upon DARA, a system designed to detect mismatches between model architecture and metadata, providing a solid foundation for enhanced security measures in the future.

Recent work has also shown that typosquatting attack, or package confusion attack, which is the most common traditional software supply chain attack, can also affect the AI model supply chain (ProtectAI 2024; Jiang et al. 2025). Our results suggest that existing techniques for typosquat detection may not apply directly to PTMs. Most existing typoquatting detection techniques rely on syntactic variations, but ignore attacks involving name semantics (Taylor et al. 2020b; Neupane et al. 2023). We conjecture that such approaches are useful for traditional packages, whose names are quite distinct and not semantically meaningful (cf. Table 1). In traditional software packages, small tweaks to a package name are a signal of typosquatting. For PTMs, our data show that small tweaks to a name are part of the naming convention for PTM names (§6.1, §6.2), PTM names contain much more semantic information and are much more structured than are the names of traditional software packages. This distinction indicates that conventional typosquatting detection approaches will detect some classes of typosquatting attacks for PTMs, but will miss attacks that leverage this property of PTM names. We suggest that techniques of **semantically-enriched** string similarity – for example, combining character-level edit distance with token-level embedding distances – and **transformer-based** anomaly classification might be applied to PTMs,

because their semantically aware classification system can group and distinguish models based on rich architectural and task metadata. However, those techniques may need to be adapted for the following aspects of PTM names: multi-part tokenization (*e.g.,*separating architecture, task, dataset and hyperparameters), domain-specific abbreviations and acronyms, versioning conventions embedded within names, and numeric parameter encodings (§7).

10 Conclusion

Pre-Trained Models (PTMs) are a new unit of reuse in software engineering, where naming practices significantly impact their reusability. Similarly to prior reuse units, engineers struggle to name PTMs well. We conducted the first study of PTM naming practices by a mixed-method approach, including a survey and a repository mining study. Our finding shows that PTM naming is different from traditional naming in terms of evolution practices and reuse decisions. We also show that engineers prefer PTM naming with architecture/function, and design purpose is important in PTM naming. PTM users look for naming inconsistencies using metadata and manual inspection. Our tool applies a novel feature extraction method and indicated that we can effectively detect naming inconsistencies with only architectural information. This study opens up several avenues for future work: establishing best practices for PTM naming, enhancing model search capabilities, and improving automated detection of naming inconsistencies. We have identified a context where new types of names are emerging, reflecting the evolving landscape of software engineering.

Author Contributions The contributions of each author are as follows: – **Wenxin Jiang:** Lead author; conducted the literature review and defined the problem scope; designed and administered the survey study and its analysis; designed DARA and all evaluation experiments. – **Mingyu Kim:** Led the implementation of naming convention extraction and manual evaluation in the repository mining study; implemented advanced feature-extraction and anomaly-detection methods within DARA; implemented all evaluation experiments and visualizations relevant to DARA. – **Chingwo Cheung:** Developed the repository-mining prototype for naming-element extraction; designed and implemented DARA's abstract architecture extraction. – **Heesoo Kim:** Co-designed and evaluated the repository mining study. – **George K. Thiruvathukal:** Provided feedback on study design, evaluation methodology, and problem statement. – **James C. Davis:** Supervised all aspects of study conduct; performed qualitative analysis of survey data (Tables 6-7); guided DARA's evaluation process.

Funding This research was supported by gifts from Google and Cisco, and by NSF awards #2107230, #2229703, #2107020, and #2104319.

Data Availability The artifact of this work is available at https://github.com/PurdueDualityLab/PTM-Naming. The artifact includes: – **Survey study (§5.1)**: We provide the complete survey instrument, along with the anonymized raw data and data analysis sheets used in the analysis. – **Repository Mining (§5.2)**: We include the prompts, evaluation and data – **DARA (§7)**: We include the end-to-end pipeline, fine-tuning code, feature extraction, visualization, and scripts to generate all figures and tables.

Declarations

Ethical Approval Our research involving human participants, specifically the survey study, was conducted under a protocol approved by Purdue University's Institutional Review Board (IRB), with protocol number *#IRB-2022-606*.

Informed Consent All participants provided written informed consent prior to taking part in our survey study. All anonymized responses are included in the published artifact.

Conflict of Interest The authors declare that they have no conflict of interest.

Clinical Trial Number Not applicable.

References

(2021) Sample size calculator. https://www.surveysystem.com/sscalc.htm

Abdalkareem R, Nourry O, Wehaibi S, Mujahid S, Shihab E (2017) Why do developers use trivial packages? an empirical case study on npm. In: European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)

Abdellatif A, Zeng Y, Elshafei M, Shihab E, Shang W (2020) Simplifying the search of npm packages. Inf Softw Technol 126:106365

Abiodun OI, Jantan A, Omolara AE, Dada KV, Mohamed NA, Arshad H (2018) State-of-the-art in artificial neural network applications: a survey. Heliyon 4(11)

Ali M, Shiaeles S, Bendiab G, Ghita B (2020) Malgra: machine learning and n-gram malware feature extraction and detection system. Electronics 9(11):1777

Allamanis M, Barr ET, Bird C, Sutton C (2015) Suggesting accurate method and class names. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering. ACM, Bergamo Italy, pp 38–49. https://doi.org/10.1145/2786805.2786849, https://dl.acm.org/doi/10.1145/2786805.2786849

Alpern R, Lazer I, Tzachor I, Hakim H, Weissbuch S, Feitelson DG (2024) Reproducing, extending, and analyzing naming experiments. arXiv

Alsuhaibani R, Newman C, Decker M, Collard M, Maletic J (2021) On the naming of methods: a survey of professional developers. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp 587–599. https://doi.org/10.1109/ICSE43902.2021.00061

Ayodele TO (2010) Types of machine learning algorithms. New Adv Mach Learn 3(19–48):5–1

Azure and contributors (2023) Abusing ml model file formats to create malware on ai systems: a proof of concept. https://github.com/Azure/counterfit/wiki/Abusing-ML-model-file-formats-to-create-malware-on-AI-systems:-A-proof-of-concept#detection

Beltagy I, Peters ME, Cohan A (2020) Longformer: the long-document transformer. arXiv:2004.05150

Bengio Y, Courville A, Vincent P (2013) Representation learning: a review and new perspectives. IEEE Trans Pattern Anal Mach Intell 35(8):1798–1828

Bhatia A, Eghan EE, Grichi M, Cavanagh WG, Jiang ZM, Adams B (2023) Towards a change taxonomy for machine learning pipelines: empirical study of ml pipelines and forks related to academic publications. Empir Softw Eng 28(3):60

Bober-Irizar M, Shumailov I, Zhao Y, Mullins R, Papernot N (2023) Architectural backdoors in neural networks. In: 2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, Vancouver, BC, Canada, pp 24595–24604. https://doi.org/10.1109/CVPR52729.2023.02356, https://ieeexplore.ieee.org/document/10204575/

Boogerd C, Moonen L (2008) Assessing the value of coding standards: an empirical study. In: 2008 IEEE International conference on software maintenance. IEEE, pp 277–286

Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A et al (2020a) Language models are few-shot learners. Adv Neural Inf Process Syst 33:1877–1901

Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A et al (2020b) Language models are few-shot learners. Adv Neural Inf Process Syst 33:1877–1901

Bubeck S, Chandrasekaran V, Eldan R, Gehrke J, Horvitz E, Kamar E, Lee P, Lee YT, Li Y, Lundberg S, Nori H, Palangi H, Ribeiro MT, Zhang Y (2023) Sparks of artificial general intelligence: early experiments with GPT-4. http://arxiv.org/abs/2303.12712
Butler S, Wermelinger M, Yu Y, Sharp H (2010) Exploring the influence of identifier names on code quality: an empirical study. In: 2010 14th European conference on software maintenance and reengineering, pp 156–165. https://doi.org/10.1109/CSMR.2010.27
Butler S, Wermelinger M, Yu Y, Sharp H (2011) Mining java class naming conventions. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp 93–102. https://doi.org/10.1109/ICSM.2011.6080776, iSSN: 1063-6773
Capra M, Bussolino B, Marchisio A, Masera G, Martina M, Shafique M (2020) Hardware and software optimizations for accelerating deep neural networks: survey of current trends, challenges, and the road ahead. IEEE Access 8:225134–225180
Castaño J, Martínez-Fernández S, Franch X, Bogner J (2023) Analyzing the evolution and maintenance of ml models on hugging face. arXiv:2311.13380
Checkmarx Research Team (2023) A new stealthier type of typosquatting attack spotted targeting npm. https://checkmarx.com/blog/a-new-stealthier-type-of-typosquatting-attack-spotted-targeting-npm/
Chmielinski KS, Newman S, Taylor M, Joseph J, Thomas K, Yurkofsky J, Qiu YC (2022) The dataset nutrition label (2nd gen): leveraging context to mitigate harms in artificial intelligence. arXiv:2201.03954
Chung J, Gulcehre C, Cho K, Bengio Y (2014) Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv:1412.3555
Davis JC, Jajal P, Jiang W, Schorlemmer TR, Synovic N, Thiruvathukal GK (2023) Reusing deep learning models: challenges and directions in software engineering. In: Proceedings of the IEEE John Vincent Atanasoff symposium on modern computing (JVA'23)
Decan A, Mens T, Grosjean P (2019) An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empir Softw Eng 24(1):381–416
DeepSeek AI (2024) Deepseek ai on hugging face. https://huggingface.co/deepseek-ai. Accessed 14 Apr 2025
Deissenboeck F, Pizka M (2006) Concise and consistent naming. Software Qual J 14:261–282
Devlin J, Chang MW, Lee K, Toutanova K (2019) Bert: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers), pp 4171–4186
Di Sipio C, Rubei R, Di Rocco J, Di Ruscio D, Nguyen PT (2024) Automated categorization of pre-trained models for software engineering: a case study with a hugging face dataset. arXiv:2405.13185
Feitelson DG, Mizrahi A, Noy N, Shabat AB, Eliyahu O, Sheffer R (2020) How developers choose names. IEEE Trans Software Eng 48(1):37–52
Frakes WB, Kang K (2005) Software reuse research: status and future. IEEE Trans Software Eng 31(7):529–536
Gao S, Chen C, Xing Z, Ma Y, Song W, Lin SW (2019) A neural model for method name generation from functional description. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 414–421. https://doi.org/10.1109/SANER.2019.8667994, iSSN: 1534-5351
Garg G (2023) The power of hugging face ai. https://medium.com/@gargg/the-power-of-hugging-face-ai-4f6558ee0874
Geisler S, Li Y, Mankowitz DJ, Cemgil AT, Günnemann S, Paduraru C (2023) Transformers meet directed graphs. In: International Conference on Machine Learning (ICML'23). PMLR, pp 11144–11172
González R, Van Der Meer K (2004) Standard metadata applied to software retrieval. J Inf Sci 30(4):300–309
Google (2024a) Google on hugging face. https://huggingface.co/google. Accessed 14 Apr 2025
Google (2024b) Google Style Guide. https://google.github.io/styleguide/
Gresta R, Durelli V, Cirilo E (2023) Naming practices in object-oriented programming: an empirical study. J Softw Eng Res Dev, pp 5–1
Griss ML (1993) Software reuse: from library to factory. IBM Syst J 32(4):548–566
Géron A (2019) Hands-on machine learning with scikit-learn, keras, and tensorflow: concepts, tools, and techniques to build intelligent systems, 2nd edn. O'Reilly Media
Gu T, Liu K, Dolan-Gavitt B, Garg S (2019) Badnets: evaluating backdooring attacks on deep neural networks. IEEE Access 7:47230–47244
Gu Y, Ying L, Pu Y, Hu X, Chai H, Wang R, Gao X, Duan H (2023) Investigating package related security threats in software registries. In: 2023 IEEE symposium on Security and Privacy (SP). IEEE, pp 1578–1595
Guest G, MacQueen KM, Namey EE (2011) Applied thematic analysis. Sage publications
Guido van Rossum et al (2001) PEP 8 – style guide for python code. https://peps.python.org/pep-0008/

Gunel B, Du J, Conneau A, Stoyanov V (2020) Supervised contrastive learning for pre-trained language model fine-tuning. arXiv:2011.01403

Gururangan S, Marasović A, Swayamdipta S, Lo K, Beltagy I, Downey D, Smith NA (2020) Don't stop pretraining: adapt language models to domains and tasks. In: Proceedings of the 58th annual meeting of the Association for Computational Linguistics (ACL)

Habchi S, Blanc X, Rouvoy R (2018) On adopting linters to deal with performance concerns in android apps. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pp 6–16

Han X, Zhang Z, Ding N, Gu Y, Liu X, Huo Y, Qiu J, Yao Y, Zhang A, Zhang L, Han W, Huang M, Jin Q, Lan Y, Liu Y, Liu Z, Lu Z, Qiu X, Song R, Tang J, Wen JR, Yuan J, Zhao WX, Zhu J (2021) Pre-trained models: past, present and future. AI Open 2:225–250

He J, Lee CC, Raychev V, Vechev M (2021a) Learning to find naming issues with big code and small supervision. In: Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation, pp 296–311

He X, Zhao K, Chu X (2021b) AutoML: a survey of the state-of-the-art. Knowl-Based Syst

Heineman GT, Councill WT (2001) Component-based software engineering. Putting the Pieces Together, Addison-Westley 5:1

Henninger S (1994) Using iterative refinement to find reusable software. IEEE Softw 11(5):48–59

Hepworth I, Olive K, Dasgupta K, Le M, Lodato M, Maruseac M, Meiklejohn S, Chaudhuri S, Minkus T (2024) Securing the ai software supply chain. Technical report, Google

HiddenLayer (2024) Ai threat landscape report. https://21998286.fs1.hubspotusercontent-na1.net/hubfs/21998286/HiddenLayer%20AI%20Threat%20Landscape%20Report%202024.pdf, retrieved from HiddenLayer

Hofmeister J, Siegmund J, Holt DV (2017a) Shorter identifier names take longer to comprehend. In: 2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER). IEEE, pp 217–227

Høst EW, Østvold BM (2009) Debugging method names. In: European conference on object-oriented programming. Springer, pp 294–317

Huang J, Lin B (2023) Cigar: contrastive learning for github action recommendation. In: 2023 IEEE 23rd international working conference on Source Code Analysis and Manipulation (SCAM). IEEE, pp 61–71

Hugging Face (2023) Hugging face documentations. https://huggingface.co/docs

Hugging Face, Inc (2024) Hugging face terms of service. https://huggingface.co/terms-of-service. Accessed 18 Aug 2024

Infosecurity Magazine (2023) Malicious npm package uses new stealthier typosquatting attack. https://www.infosecurity-magazine.com/news/malicious-npm-package-uses/

Islam S, Elmekki H, Elsebai A, Bentahar J, Drawel N, Rjoub G, Pedrycz W (2023) A comprehensive survey on applications of transformers for deep learning tasks. Expert Syst Appl, p 122666

Jain A, Verma Y (2021) Contrastive learning of visual-semantic embeddings. arXiv:2110.08872

Jajal P, Jiang W, Tewari A, Kocinare E, Woo J, Sarraf A, Lu YH, Thiruvathukal GK, Davis JC (2024) Interoperability in deep learning: a user survey and failure analysis of onnx model converters. The 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'24)

Jiang W, Synovic N, Sethi R, Indarapu A, Hyatt M, Schorlemmer TR, Thiruvathukal GK, Davis JC (2022a) 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, pp 105–114

Jiang W, Synovic N, Sethi R, Indarapu A, Hyatt M, Schorlemmer TR, Thiruvathukal GK, Davis JC (2022b) An empirical study of artifacts and security risks in the pre-trained model supply chain. In: ACM workshop on software Supply Chain Offensive Research and Ecosystem Defenses (SCORED'22), pp 105–114. https://doi.org/10.1145/3560835.3564547

Jiang W, Banna V, Vivek N, Goel A, Synovic N, Thiruvathukal GK, Davis JC (2023a) Challenges and practices of deep learning model reengineering: a case study on computer vision. arXiv:2303.07476

Jiang W, Synovic N, Hyatt M, Schorlemmer TR, Sethi R, Lu YH, Thiruvathukal GK, Davis JC (2023b) An empirical study of pre-trained model reuse in the hugging face deep learning model registry. In: IEEE/ACM 45th International Conference on Software Engineering (ICSE'23)

Jiang W, Synovic N, Jajal P, Schorlemmer TR, Tewari A, Pareek B, Thiruvathukal GK, Davis JC (2023c) Ptmtorrent: a dataset for mining open-source pre-trained model packages. Proceedings of the 20th international conference on Mining Software Repositories (MSR'23)

Jiang W, Yasmin J, Jones J, Synovic N, Kuo J, Tian Y, Thiruvathukal GK, Davis JC (2024) Peatmoss: a dataset and initial analysis of pre-trained models in open-source software. In: Proceedings of the 21th annual conference on Mining Software Repositories (MSR'24)

Jiang W, Çakar B, Lysenko M, Davis JC (2025) Detecting active and stealthy typosquatting threats in package registries. arXiv:2502.20528
Johnson RB, Onwuegbuzie AJ (2004) Mixed methods research: a research paradigm whose time has come. Educational Researcher
Jones J, Jiang W, Synovic N, Thiruvathukal GK, Davis JC (2024) What do we know about hugging face? a systematic literature review and quantitative validation of qualitative claims. In: Proceedings of the 18th international conference on Empirical Software Engineering and Measurement (ESEM)
Kathikar A, Nair A, Lazarine B, Sachdeva A, Samtani S (2023) Assessing the vulnerabilities of the open-source Artificial Intelligence (AI) landscape: a large-scale analysis of the hugging face platform
Khalid H, Zimányi E (2024) Repairing raw metadata for metadata management. Inf Syst 122:102344
Khosla P, Teterwak P, Wang C, Sarna A, Tian Y, Isola P, Maschinot A, Liu C, Krishnan D (2021) Supervised contrastive learning. https://arxiv.org/abs/2004.11362, 2004.11362
Kim Y (2014) Convolutional neural networks for sentence classification. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp 1746–1751
Kitchenham BA, Pfleeger SL (2008) Personal opinion surveys. In: Guide to advanced empirical software engineering. Springer, pp 63–92
Kotonya G, Lee J (2014) Teaching reuse-driven software engineering through innovative role playing. In: Companion proceedings of the 36th international conference on software engineering, pp 276–282
Krueger CW (1992) Software reuse. ACM Comp Surv (CSUR) 24(2):131–183
Langford H, Shumailov I, Zhao Y, Mullins R, Papernot N (2024) Architectural neural backdoors from first principles. arXiv:2402.06957
Lau KK (2006) Software component models. In: Proceedings of the 28th international conference on software engineering, pp 1081–1082
Lawrie D, Morrell C, Feild H, Binkley D (2006) What's in a name? a study of identifiers. In: 14th IEEE international conference on program comprehension (ICPC'06). IEEE, pp 3–12
Lawrie D, Morrell C, Feild H, Binkley D (2007) Effective identifier names for comprehension and memory. Innovations Syst Softw Eng 3(4):303–318. https://doi.org/10.1007/s11334-007-0031-2, http://link.springer.com/10.1007/s11334-007-0031-2
Le VH, Zhang H (2022) Log-based anomaly detection with deep learning: how far are we? In: Proceedings of the 44th international conference on software engineering, pp 1356–1367
Le Scao T, Fan A, Akiki C, Pavlick E, Ilić S, Hesslow D, Castagné R, Luccioni AS, Yvon F, Gallé M, et al. (2022) Bloom: a 176b-parameter open-access multilingual language model
LeCun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521(7553):436–444. https://doi.org/10.1038/nature14539
Lester B, Al-Rfou R, Constant N (2021) The power of scale for parameter-efficient prompt tuning. http://arxiv.org/abs/2104.08691
Li J, Tang T, Zhao WX, Nie JY, Wen JR (2024) Pre-trained language models for text generation: a survey. ACM Comput Surv 56(9):1–39
Li Z, Hai R, Bozzon A, Katsifodimos A (2022) Metadata representations for queryable ML model zoos. http://arxiv.org/abs/2207.09315, arXiv:2207.09315
Liu H, Liu Q, Staicu CA, Pradel M, Luo Y (2016) Nomen est omen: exploring and exploiting similarities between argument and parameter names. In: Proceedings of the 38th international conference on software engineering, pp 1063–1073
Liu S, Demirel MF, Liang Y (2019a) N-gram graph: simple unsupervised representation for graphs, with applications to molecules. Adv Neural Inf Process Syst 32
Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, Levy O, Lewis M, Zettlemoyer L, Stoyanov V (2019b) Roberta: a robustly optimized bert pretraining approach. arXiv:1907.11692
Lu B, Yang J, Chen LY, Ren S (2019) Automating deep neural network model selection for edge inference. In: 2019 IEEE first international conference on Cognitive Machine Intelligence (CogMI), pp 184–193. https://doi.org/10.1109/CogMI48466.2019.00035
Ma H, Rong Y, Huang J (2022) Graph neural networks: scalability. Graph neural networks: foundations, frontiers, and applications, pp 99–119
Melegati J, Wang X, Abrahamsson P (2019) Hypotheses engineering: first essential steps of experiment-driven software development. In: 2019 IEEE/ACM joint 4th international workshop on rapid continuous software engineering and 1st international workshop on data-driven decisions, experimentation and evolution (RCoSE/DDrEE). IEEE, pp 16–19
Meta AI (2024) Meta ai on hugging face. https://huggingface.co/facebook. Accessed 14 Apr 2025
Montes D, Peerapatanapokin P, Schultz J, Guo C, Jiang W, Davis JC (2022) Discrepancies among pre-trained deep neural networks: a new threat to model zoo reliability. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE-IVR track)

Neupane S, Holmes G, Wyss E, Davidson D, De Carli L (2023) Beyond typosquatting: an in-depth look at package confusion. In: 32nd USENIX security symposium (USENIX security 23), pp 3439–3456
Nguyen G, Islam MJ, Pan R, Rajan H (2022) Manas: mining software repositories to assist AutoML. In: International Conference on Software Engineering (ICSE). ACM, Pittsburgh Pennsylvania, pp 1368–1380. https://doi.org/10.1145/3510003.3510052, https://dl.acm.org/doi/10.1145/3510003.3510052
Ni R, Shu M, Souri H, Goldblum M, Goldstein T (2021) The close relationship between contrastive learning and meta-learning. In: International Conference on Learning Representations (ICLR'21)
NPM (2023) Package name guidelines. https://docs.npmjs.com/package-name-guidelines
NVIDIA (2024) Inconsistent model config and architecture - issue #10006. https://github.com/NVIDIA/NeMo/issues/10006. Accessed 14 Apr 2025
Ohm M, Plate H, Sykosch A, Meier M (2020) Backstabber's knife collection: a review of open source software supply chain attacks. In: Maurice C, Bilge L, Stringhini G, Neves N (eds) Detection of intrusions and malware, and vulnerability assessment, vol 12223. Springer International Publishing, Cham, pp 23–43. http://link.springer.com/10.1007/978-3-030-52683-2_2
Olsson HH, Bosch J (2014) From opinions to data-driven software r &d: a multi-case study on how to close the'open loop'problem. In: 2014 40th EUROMICRO conference on software engineering and advanced applications. IEEE, pp 9–16
ONNX (2023) Onnx model zoo. https://github.com/onnx/models
OpenBMB (2023) Issue #196 - project author team stay tuned: I found out that the llama3-v project is stealing a lot of academic work from minicpm-llama3-v 2.5. https://github.com/OpenBMB/MiniCPM-V/issues/196, miniCPM-V. GitHub
Patil PV, Jiang W, Peng H, Lugo D, Kalu KG, LeBlanc J, Smith L, Heo H, Aou N, Davis JC (2025) Recommending pre-trained models for iot devices. In: Proceedings of the international workshop on Software Engineering Research in Practice (SERIP), to appear
Peng B (2021) About the model name. https://github.com/allenai/longformer/issues/185
Pradel M, Gross TR (2011) Detecting anomalies in the order of equally-typed method arguments. In: Proceedings of the 2011 international symposium on software testing and analysis, pp 232–242
Pradel M, Sen K (2018) Deepbugs: a learning approach to name-based bug detection. Proc ACM Program Lang 2(OOPSLA):1–25
Preferred Networks (2021) How computational graphs are constructed in pytorch. https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/
ProtectAI (2024) Unveiling ai supply chain attacks on hugging face. https://protectai.com/threat-research/unveiling-ai-supply-chain-attacks-on-hugging-face
Pytorch (2021) Pytorch hub. https://pytorch.org/hub/
Qi B, Sun H, Gao X, Zhang H, Li Z, Liu X (2023) Reusing deep neural network models through model re-engineering. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, pp 983–994
Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I (2019) Language models are unsupervised multitask learners. OpenAI blog 1(8):9
Ren H, Xu B, Wang Y, Yi C, Huang C, Kou X, Xing T, Yang M, Tong J, Zhang Q (2019) Time-series anomaly detection service at microsoft. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining, pp 3009–3017
ReversingLabs Threat Research Team (2023) r77 rootkit typosquatting: Npm threat research. https://www.reversinglabs.com/blog/r77-rootkit-typosquatting-npm-threat-research
Rogers A, Kovaleva O, Rumshisky A (2021) A primer in bertology: what we know about how bert works. Trans Assoc Comput Linguistics 8:842–866
Sabor KK, Hamou-Lhadj A, Larsson A (2017) Durfex: a feature extraction technique for efficient detection of duplicate bug reports. In: 2017 IEEE international conference on software quality, reliability and security (QRS). IEEE, pp 240–250
Saieva A, Chakraborty S, Kaiser G (2023) On contrastive learning of semantic similarity forcode to code search. arXiv:2305.03843
Saldaña J (2021) The coding manual for qualitative researchers. SAGE publications Ltd
Schelter S, Biessmann F, Januschowski T, Salinas D, Seufert S, Szarvas G (2018) On challenges in machine learning model management. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering
Seacord RC, Hissam SA, Wallnau KC (1998) Agora: a search engine for software components. IEEE Internet Comput 2(6):62
Sejfia A, Schäfer M (2022) Practical automated detection of malicious npm packages. In: Proceedings of the 44th international conference on software engineering, pp 1681–1692
Sens Y, Knopp H, Peldszus S, Berger T (2024) A large-scale study of model integration in ml-enabled software systems. arXiv:2408.06226

shersoni610, Li C (2024) Issue #1136: naming llava models. https://github.com/haotian-liu/LLaVA/issues/1136
Sohacheski DB, Lurie Y, Mark S (2021) Software identifier naming conventions & dictionary. WSEAS Trans Comput Res 9:21–32
Sommerville I (2015) Software engineering, 10th edn. Pearson Education Limited
Soto-Valero C, Benelallam A, Harrand N, Barais O, Baudry B (2019) The emergence of software diversity in maven central. In: 2019 IEEE/ACM 16th international conference on Mining Software Repositories (MSR). IEEE, pp 333–343
Storey MA, Hoda R, Milani AMP, Baldassarre MT (2025) Guiding principles for using mixed methods research in software engineering. Empir Softw Eng (EMSE)
Sutter H, Alexandrescu A (2004) C++ coding standards: 101 rules, guidelines, and best practices. Pearson Education
Szyperski C, Gruntz D, Murer S (2002) Component software: beyond object-oriented programming, 2nd edn. Addison-Wesley
Tan X, Li T, Chen R, Liu F, Zhang L (2024) Challenges of using pre-trained models: the practitioners' perspective. arXiv:2404.14710
Taraghi M, Dorcelus G, Foundjem A, Tambon F, Khomh F (2024) Deep learning model reuse in the huggingface community: challenges, benefit and trends. arXiv:2401.13177
Taye MM (2023) Theoretical understanding of convolutional neural network: concepts, architectures, applications, future directions. Computation 11(3):52
Taylor M, Vaidya R, Davidson D, De Carli L, Rastogi V (2020a) Defending against package typosquatting. In: Kutyłowski M, Zhang J, Chen C (eds) Network and system security. Springer International Publishing, Cham. Lecture Notes in Computer Science, pp 112–131. https://doi.org/10.1007/978-3-030-65745-1_7
Taylor M, Vaidya R, Davidson D, De Carli L, Rastogi V (2020b) Defending against package typosquatting. In: Network and system security: 14th international conference, NSS 2020, Melbourne, VIC, Australia, November 25–27, 2020, Proceedings 14. Springer, pp 112–131
TensorFlow (2023) Tensorflow hub. https://www.tensorflow.org/hub
Terdchanakul P, Hata H, Phannachitta P, Matsumoto K (2017) Bug or not? bug report classification using n-gram idf. In: 2017 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 534–538
Tómasdóttir KF, Aniche M, Van Deursen A (2018) The adoption of javascript linters in practice: a case study on eslint. IEEE Trans Software Eng 46(8):863–891
Tsay J, Braz A, Hirzel M, Shinnar A, Mummert T (2020) Aimmx: artificial intelligence model metadata extractor. In: Proceedings of the 17th international conference on mining software repositories, pp 81–92
Vassallo C, Proksch S, Jancso A, Gall HC, Di Penta M (2020) Configuration smells in continuous delivery pipelines: a linter and a six-month study on gitlab. In: Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 327–337
Vaswani A (2017) Attention is all you need. Adv Neural Inf Process Syst
Verdecchia R, Engström E, Lago P, Runeson P, Song Q (2023) Threats to validity in software engineering research: a critical reflection. Inf Softw Technol 164:107329
Violos J, Tserpes K, Varlamis I, Varvarigou T (2018) Text classification using the n-gram graph representation model over high frequency data streams. Front Appl Math Stat 4:41
Wang C, Chen Z, Zhou M (2023a) AutoML from software engineering perspective: landscapes and challenges. In: 2023 IEEE/ACM 20th international conference on Mining Software Repositories (MSR). IEEE, Melbourne, Australia, pp 39–51. https://doi.org/10.1109/MSR59073.2023.00019, https://ieeexplore.ieee.org/document/10173951/
Wang S, Manning CD (2012) Baselines and bigrams: simple, good sentiment and topic classification. In: Proceedings of the 50th annual meeting of the Association for Computational Linguistics (ACL), pp 90–94
Wang S, Wen M, Lin B, Liu Y, Bissyandé TF, Mao X (2023b) Pre-implementation method name prediction for object-oriented programming. ACM Trans Softw Eng Method (TOSEM). https://doi.org/10.1145/3597203, https://dl.acm.org/doi/10.1145/3597203
Wang Z, Liu C, Cui X, Yin J, Wang X (2022) EvilModel 2.0: bringing neural network models into malware attacks. Comput Secur. https://doi.org/10.1016/j.cose.2022.102807
Winkler J, Agarwal A, Tung C, Ugalde DR, Jung YJ, Davis JC (2021) A replication of 'deepbugs: a learning approach to name-based bug detection'. In: Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 1604–1604
Wittern E, Suter P, Rajagopalan S (2016) A look at the dynamics of the JavaScript package ecosystem. In: International conference on Mining Software Repositories (MSR)

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A et al (2012) Experimentation in software engineering, vol 236. Springer

Wu G, Cao Y, Chen W, Wei J, Zhong H, Huang T (2017) Appcheck: a crowdsourced testing service for android applications. In: 2017 IEEE International Conference on Web Services (ICWS). IEEE, pp 253–260

Wu Z, Pan S, Chen F, Long G, Zhang C, Philip SY (2020) A comprehensive survey on graph neural networks. IEEE Trans Neural Netw Learn Syst 32(1):4–24

You K, Liu Y, Wang J, Jordan MI, Long M (2021a) Ranking and tuning pre-trained models: a new paradigm of exploiting model hubs. J Mach Learn Res (JMLR) 23(1):9400–9446. http://arxiv.org/abs/2110.10545

You K, Liu Y, Wang J, Long M (2021b) LogME: practical assessment of pre-trained models for transfer learning. In: International Conference on Machine Learning (ICML). PMLR, pp 12133–12143. https://proceedings.mlr.press/v139/you21b.html

Zaigrajew V, Zieba M (2022) Contrastive learning for multi-label classification. In: Proceedings of conference on neural information processing systems, New Orleans, pp 1–8

Zamfirescu-Pereira J, Wong RY, Hartmann B, Yang Q (2023) Why johnny can't prompt: how non-ai experts try (and fail) to design llm prompts. In: Proceedings of the 2023 CHI conference on human factors in computing systems, pp 1–21

Zhang X, Zhao J, LeCun Y (2015) Character-level convolutional networks for text classification. In: Advances in Neural Information Processing Systems (NeurIPS)

Zhang YK, Huang TJ, Ding YX, Zhan DC, Ye HJ (2024) Model spider: learning to rank pre-trained models efficiently. Adv Neural Inf Process Syst 36

Zhao J, Wang S, Zhao Y, Hou X, Wang K, Gao P, Zhang Y, Wei C, Wang H (2024) Models are codes: towards measuring malicious code poisoning attacks on pre-trained model hubs. In: Proceedings of the 39th IEEE/ACM international conference on automated software engineering, pp 2087–2098

Zhu P, Zuo W, Zhang L, Hu Q, Shiu SC (2015) Unsupervised feature selection by regularized self-representation. Pattern Recogn 48(2):438–446

Authors and Affiliations

Wenxin Jiang[1] · Mingyu Kim[1] · Chingwo Cheung[1] · Heesoo Kim[1] · George K. Thiruvathukal[2] · James C. Davis[1]

✉ Wenxin Jiang
jiang784@purdue.edu

✉ James C. Davis
davisjam@purdue.edu

Mingyu Kim
kim3118@purdue.edu

Chingwo Cheung
cheung59@purdue.edu

Heesoo Kim
kim2903@purdue.edu

George K. Thiruvathukal
gkt@cs.luc.edu

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

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