Noname manuscript No.
(will be inserted by the editor)

On the synchronization between Hugging Face pre-trained language models and their upstream GitHub repository

Ajibode Adekunle · Abdul Ali Bangash · Bram Adams · Ahmed E. Hassan

Received: date / Accepted: date

Abstract Pre-trained language models (PTLMs) have revolutionized the field of natural language processing (NLP), enabling significant advancements in tasks such as text generation and translation. Similar to software package management, which involves centralized version control and distributed consumption, PTLMs are trained using code and environment scripts hosted in an upstream repository (e.g., a GitHub (GH) repository), while the family of model variants trained from a given repository's scripts are distributed using dedicated downstream distribution platforms like Hugging Face (HF). Despite these similarities, coordinating development activities between GH and HF presents several challenges to avoid misaligned release timelines, inconsistent versioning practices, and other obstacles to seamless reuse of PTLM variants. To understand how commit activities are coordinated between these two platforms, we conducted an in-depth mixed-method study of 325 PTLM families consisting of 904 HF PTLM variants. Our analysis reveals that GH contributors typically make changes related to specifying the version of the model, improving code quality, performance optimization, and dependency management within the training scripts, while HF contributors make changes related to improving model descriptions, data set handling, and setup required for model inference. Furthermore, to understand the synchronization aspects of commit activities between GH and HF, we examined three dimensions of these activities—lag (delay), type of synchronization, and intensity—which together yielded eight distinct synchronization patterns. The prevalence of partially synchronized patterns, such as *Disperse synchronization* and *Sparse synchronization*, reveals structural disconnects in current cross-platform release practices. These patterns often result in isolated changes—where improvements or fixes made on one platform are never replicated on the other—and in some cases, indicate an abandonment of one repository in favor of the other. Such fragmentation risks exposing end users to incomplete, outdated, or behaviorally inconsistent models. Hence, recognizing these synchronization patterns is critical for improving oversight and traceability in PTLM release workflows.

Adekunle Ajibode
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: ajibode.a@queensu.ca

Abdul Ali Bangash
SBASSE, Lahore University of Management Sciences, Lahore, Pakistan
E-mail: abdulali@lums.edu.pk

Bram Adams
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: bram.adams@queensu.ca

Ahmed E. Hassan
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: hassan@queensu.ca

Keywords Pre-trained Language Models, Synchronization Patterns, Commit Change Types, Coordination, Upstream, Downstream

1 Introduction

In recent years, pre-trained language models (PTLMs) have become integral to the development of Natural Language Processing (NLP) systems, due to their effectiveness in improving performance and reducing the need for task-specific training. These models have driven significant progress in tasks such as text generation, translation, and sentiment analysis (Min et al., 2023). Beyond research, PTLMs now influence software engineering practices by powering intelligent applications like chatbots and code assistants, and automating tasks such as documentation generation, bug reporting, and code summarization (Hou et al., 2024). Their ability to capture the structure and semantics of human language makes them particularly effective for building intelligent systems from large text data (Wang et al., 2022). As a result, PTLMs have become foundational to modern AI development. Their distribution is facilitated by several public model repositories, including *HF*[1], *ONNX*[2], *PyTorch Hub*[3], *Model-Zoo*[4], and *Modelhub*[5], which enhance accessibility and promote the reuse of models across diverse applications (Zhao et al., 2023).

Among these platforms, HF stands out not only for hosting model weights but also for offering tools for fine-tuning and deployment (Castaño et al., 2024b), including metadata, tokenizer files, inference scripts, and documentation, which significantly simplify adoption and integration. In contrast, platforms like GH serve as the core infrastructure for collaborative development and version control, with repositories typically containing training code, preprocessing scripts, experiment configurations, and usage documentation. Yet, the interaction between both platforms is not straightforward, as our preliminary exploration in Section 3 reveals that multiple PTLMs—regardless of whether they share the same base model, such as BERT (Devlin et al., 2018), GPT (OpenAI, 2023), and RoBERTa (Liu et al., 2019)—are often developed and maintained within a single GH repository. To reflect this structure, we introduce the term 'PTLM family', which refers to a group of PTLMs managed within the same upstream GH repository.

Within a PTLM family, the roles of the upstream GH and downstream HF repositories are similar to those of open-source projects and Linux distributions, respectively, with the former performing core development activities such as creating features, resolving bugs, and implementing improvements while the latter phase focus on packaging, distributing, and preparing models for deployment. In the context of PTLMs development, this lifecycle is part of a broader and more intricate machine learning (ML) supply chain, which encompasses not only software components but also the interdependencies among datasets, model reuse, fine-tuning, and other iterative adaptation processes that make modern ML systems increasingly complex and non-linear (Stalnaker et al., 2025).

Despite the similarity with traditional open-source ecosystems, the coordination of development activities of PTLMs between GH as the upstream platform and HF as the downstream platform can present unique challenges. Coordination involves managing interdependencies between tasks, aligning goals, negotiating responsibilities, and maintaining shared understanding across individuals and teams (Kraut and Streeter, 1995). Since coordination is a complex phenomenon, this study, as a first step, focuses specifically on the synchronization aspect of PTLM release activity coordination. Synchronization is an observable and measurable form of coordination, providing a lower bound on the overall coordination effort. Specifically, in this study, we refer synchronization as the simultaneous occurrence of related activities on both GH and HF.

The existing problem on synchronizing related activities between upstream GH and downstream HF is not just theoretical; it manifests in real-world issues encountered in PTLM projects. One such example is the PTLM project cahya/bert-base-indonesian-522M, which demonstrates this disconnect. The HF repository was

[1] https://huggingface.co/models

[2] https://github.com/onnx/models

[3] https://pytorch.org/hub/

[4] https://modelzoo.co/

[5] http://app.modelhub.ai/

created on June 23, 2020, while the corresponding GitHub repository had been in existence since August 19, 2018. It is likely that the model files were initially transferred from GH. However, after June 23, 2020, activities ceased on HF until September 22, 2020, despite multiple updates occurring on GH throughout July and August. During this period, numerous commits on GH involved updates to the model card, changes to the base model in the codebase, corrections to model names, and the addition of a training script. When activities resumed on HF, only a few updates—mainly related to essential configuration files and converted model weight files—were replicated from GH. Consequently, some model information on HF remained outdated or incomplete, indicating a gap in consistency between the two platforms. This example illustrates how developers may update training code, configurations, or other components on GH that might alter the model's behavior, but fail to propagate these changes to HF. Such a disconnect can result in users accessing outdated or inconsistent model versions, thereby undermining reproducibility, trust, and deployment reliability.

Despite the practical importance of synchronizing cross-platform development activities, this topic remains underexplored in the context of PTLMs, even as its dynamics differ significantly from those in traditional software engineering. For instance, ongoing community discussions on HF[6][7] reveal persistent challenges in synchronizing upstream GitHub and downstream HF repositories. Specifically, little is known about the types of changes reflected in commit activities across GH and HF in PTLM development, the timing patterns of such synchronization, and how these patterns vary and evolve across PTLM families of different ages.

To address this gap, our study empirically investigates the nature, types, and frequency of changes made across HF and GH as well as how these activities are synchronized in time. This work presents the first systematic analysis of cross-platform commit activity synchronization between the two platforms. Our findings offer insights to help stakeholders understand current synchronization behaviors, identify existing gaps, and explore opportunities to improve model release workflows. By characterizing these practices, the study raises awareness among researchers and practitioners and lays the groundwork for future efforts toward more reliable and better-synchronized model development processes across platforms. Specifically, we address the following research questions:

RQ_1. **How do the changes in the commit activities of upstream and downstream repositories of PTLMs compare to each other?**

Motivation: Understanding the relationship between commits on GH and HF—specifically whether a clear upstream-downstream dynamic exists between the two platforms, is important. To address this, we analyze the prevalent change topics in the commit activities of PTLMs on GH as the upstream and HF as the downstream, identify the dominant change types in commit activities across both platforms, and compare differences in the distribution of similarity in these change types across PTLM families. These insights can raise awareness among practitioners regarding the nature of changes they make and how GH changes can affect HF behavior. This understanding can help them design better methods to manage these changes, preparing them to enhance and maintain models more effectively, leveraging collaborative insights and iterative development practices.

Findings: Our analysis reveals that commit change topics of the upstream GH repositories, emphasize PTLM version specification, code quality, performance optimization, and dependency management. In contrast, commits of the downstream HF, focus on repository setup, model descriptions, dataset handling, and inference setup—reflecting the distinct roles these platforms play in the PTLM release pipeline. At a higher level of abstraction, GH commits center more on model structure, external documentation, and training infrastructure, while HF emphasizes external documentation, preprocessing, and project metadata. These differences are statistically significant and tend to evolve with model maturity. Furthermore, we find that the presence of cross-platform contributors is associated with a higher degree of similarity in change types between GH and HF repositories. This relationship appears to be influenced by model maturity and the extent of collaborative involvement.

RQ_2. **What are the synchronization patterns of commit activities across PTLMs on upstream and downstream?**

[6] https://discuss.huggingface.co/t/github-repo-and-hugging-face-repo-sync/114697

[7] https://github.com/huggingface/huggingface_hub/issues/534

Motivation: Understanding how commit activities of PTLM families synchronize over time across GH and HF platforms is crucial for uncovering current synchronization practices and identifying potential gaps or inefficiencies. This understanding enables practitioners to reflect on existing challenges and consider improvements. To address this, we investigate the underlying characteristics that shape synchronization as well as different synchronization patterns of activities between GH and HF. These insights can support the development of more efficient and aligned release workflows across platforms.
Findings: We identified *lag* (the order in which PTLM family commit activities first appear between GH and HF), *synchronization type* (how commit activities on GH and HF align over time) and *Intensity* (the frequency and concentration of commit activities across ecosystems) as key elements characterizing commit activity synchronization between GH and HF within PTLM families. Based on these characteristics, we uncovered eight distinct synchronization patterns: *Rare*, *Intermittent*, *Frequent*, *Disperse*, *Sparse*, *Dense Partial*, *Sporadic Disjoint*, and *Rare Disjoint*. These patterns reflect varying degrees and types of synchronization between upstream (GH) and downstream (HF) activities, highlighting how existing practices often lack synchronization and may lead end users to access stale or outdated models.

RQ_3. **How are the synchronization patterns between upstream and downstream distributed across PTLM families?**
Motivation: Understanding the distribution of change types within synchronization patterns and the variations of these synchronization patterns reveals how consistently updates are managed across platforms and highlights patterns of delay, alignment, and divergence between upstream and downstream activities. To address this, we first examine how different change types are distributed across synchronization patterns to determine whether the prevalent activities in each pattern differ from those observed in RQ1. This is important to understand if change types actually define the observed synchronization patterns. Furthermore, we investigate the prevalence of lag and synchronization patterns across PTLM families at different stages of maturity. We also analyze how long it typically takes for change types made on one platform to be reflected on the other, depending on the synchronization pattern and the maturity stage of the model family. This understanding can help open-source communities and end-users plan and implement more effective release and maintenance strategies as their PTLMs evolve.
Findings: Our results show that the most prevalent synchronization pattern is *Disperse* (39.4%), where activities on the two platforms occur with limited overlap and extended delay—often resulting in one platform continuing development while the other remains inactive. The distribution of change types across these synchronization patterns differ significantly, indicating that the synchronization pattern is correlated with the nature of changes made. We also observed that commit activities between platforms are often misaligned: on average, changes made on GH are reflected on HF after a lag of 15.82 days. Although contributor count correlates with increased activity across both platforms, it does not consistently lead to tightly synchronized updates, highlighting the complexity of managing cross-platform collaboration in evolving PTLM families.

Our findings highlight substantial variability in how commit activities are synchronized between GH and HF, revealing inefficiencies in synchronizing model development and release. These synchronization gaps suggest that project maintainers often lack structured workflows to manage changes across platforms, relying instead on ad hoc[8] updates. The observed lag and divergence in change topics higlight the need for improved release practices—such as lightweight automation scripts, contribution templates, or workflow policies—that explicitly support synchronization across both development and deployment stages.

Importantly, our analysis of synchronization captures only the potential temporal co-occurrence of upstream and downstream changes, rather than tracking exact change propagation across platforms. As such, the measured lags and topic divergences represent a lower bound on the true coordination gap, suggesting even greater misalignment at deeper levels of the release process. While HF provides robust tools for inference integration and model packaging, there is limited support for the kind of upstream/downstream synchronization required for end-to-end PTLM releases. Practitioners could benefit from release engineering practices that

[8]The dominance of *Disperse* and *Rare* patterns, combined with inconsistent lag times and limited cross-platform contributor overlap, suggests that synchronization between GH and HF is often handled in an ad hoc rather than systematic manner.

unify GH training pipelines with HF deployment workflows—especially as projects mature and contributor engagement evolves. Specifically, our study provides the following contributions:

- We pioneer the empirical study of the relationship between GH (as the upstream platform) and HF (as the downstream platform) in the context of pre-trained language model (PTLM) development.
- We identify key characteristics—such as lag, synchronization type, and Intensity—that shape eight distinct synchronization patterns in cross-platform model development. These patterns reveal that PTLM practitioners often rely on unstructured, ad hoc synchronization strategies. This points to the need for automated mechanisms to support more effective synchronization and release engineering workflows.
- We provide a publicly available dataset and replication package to support future research on cross-platform development synchronization and release practices within the PTLM ecosystem (Adekunle, 2025).

This paper is structured as follows. Section 2 discuss key concepts such as pre-trained language models, GH as upstream and HF as downstream for PTLM management, coordination and synchronization of development activities in PTLMs, and related work. Section 3 outlines the study setup. Section 4 presents the findings of the research questions. Section 5 covers the study's discussion and implications, while Section 6 addresses potential threats to validity. Finally, Section 7 summarizes the study and outlines key directions for future research.

2 Background and Related Work

2.1 Pre-Trained Language Models

Pre-trained large models are general-purpose models trained on large-scale datasets to extract transferable patterns, allowing fine-tuning and other adaptions for specific downstream tasks. Unlike simpler models like logistic regression, built from scratch for specific objectives, modern deep learning architectures use pre-training to capture broad data representations across domains like image recognition, speech processing, and NLP. This improvement in adaptability and performance results from combining high-capacity architectures, large datasets, and refined training methods (Mao, 2020). Within this category, PTLMs—such as BERT (Devlin et al., 2018), GPT (OpenAI, 2023), and RoBERTa (Liu et al., 2019)—are specialized for NLP tasks, using large-scale textual data to support token prediction, sequence understanding, and semantic representation. These models power NLP applications like text classification, machine translation, and question answering. This study focuses on the interaction of GitHub and Hugging Face during the development of these PTLMs.

2.2 GitHub as Upstream and Hugging Face as Downstream for PTLM Management

In software development, the upstream stage refers to the ongoing phases of development, where the core elements are designed, developed, and maintained. Upstream encompasses the design and planning of core assets such as source code, patches, and bug fixes, which are posted for collaborative action and future development within code repositories (Adrian, 2016). Downstream, conversely, refers to the projects in a supply chain that depend on or use upstream projects. The upstream and downstream roles can be illustrated within the framework of a Linux distribution, which typically comprises the kernel, a set of packages, and a package management system. In this ecosystem, an open-source distribution patches, builds, and distributes upstream projects as packages to its users (like Fedora) or external parties (like Debian) (Lin et al., 2022). This is similar to how HF trains and provides access to models whose scripts and related assets are developed upstream.

GH and HF serve distinct but complementary roles in the development and distribution of PTLMs. GH typically hosts model source code, training scripts, datasets, configuration files, and supports collaborative practices such as version control, issue tracking, and pull requests (Loeliger and McCullough, 2012, Dabbish et al., 2012). HF, on the other hand, focuses on the distribution, fine-tuning, and deployment of models. It offers APIs, model hubs, and libraries that facilitate the reuse and adaptation of models for downstream tasks. Based on the types of artifacts and activities observed on each platform for PTLMs, we consider GH the

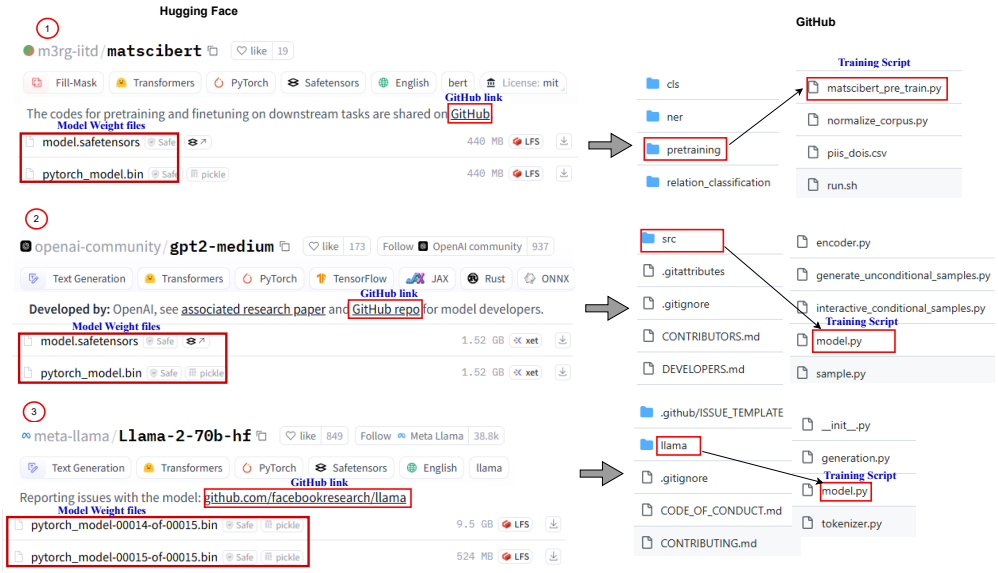

Fig. 1 Examples of PTLM GH repositories and their corresponding HF counterparts

upstream platform—where foundational development and synchronization occur—and HF the downstream platform—where models are published, reused, and integrated into applications.

To the best of our knowledge, there is no off-the-shelf tool to automate synchronization between the platforms, often leaving practitioners to manually coordinate updates across the two—an issue that surfaces in ongoing community discussions highlighted in the introduction. This highlights the need to better understand current synchronization practices and how they can be improved. As illustrated in Figure 1, many PTLMs link their HF model repositories to GH repositories, which contain key resources such as training code, datasets, and example scripts. For instance, the model m3rg-iitd/matscibert[9] notes that its code is hosted at github.com/m3RG-IITD/MatSciBERT; openai-community/gpt2-medium[10] notes github.com/openai/gpt-2; and meta-llama/Llama-2-70b-hf[11] directs users to report issues at github.com/facebookresearch/llama.

Our study examines this upstream–downstream interplay, aiming to better understand how PTLMs evolve across both environments. We also highlight the current synchronization practices involved and suggest opportunities for improving synchronization between model development and distribution workflows.

2.3 Coordination and Synchronization of Development Activities in PTLMs

In software engineering, coordination refers to the process by which individuals working on a shared project align their efforts toward common goals—agreeing on software definitions, sharing information, meshing activities, and ensuring that components integrate efficiently and without redundancy (Kraut and Streeter, 1995). Studying the coordination of development activities in PTLM ecosystems is particularly challenging due to the diversity and scope of activities involved. PTLM variants are not limited to model code alone—they also rely on interconnected components such as datasets, evaluation scripts, training configurations, and documentation.

[9] https://huggingface.co/m3rg-iitd/matscibert
[10] https://huggingface.co/openai-community/gpt2-medium
[11] https://huggingface.co/meta-llama/Llama-2-70b-hf

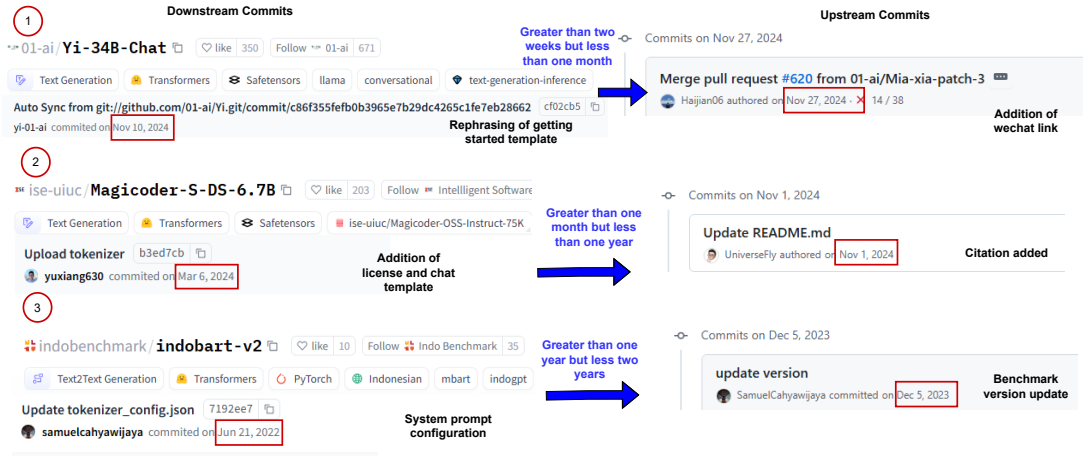

Fig. 2 Examples of delays and inconsistencies in synchronizing development activities between upstream and downstream repositories across different PTLM families. Each example shows the most recent commit on each platform at the time of data collection.

These components form a loosely integrated supply chain (Wang et al., 2025) involving both model and dataset ecosystems, making it difficult to trace how changes propagate across platforms and artifacts.

As a first step, we focus on a lower bound of coordination: *synchronization*, which considers only the temporal co-occurrence of activities on both platforms, regardless of their content. Synchronization refers to the alignment of commit activities in time to ensure consistency and minimize conflicts. In the context of PTLM development, we define synchronization as the time-based overlap of commit activities between GH and HF—serving as a proxy for whether updates occur within the same timeframe and in a coordinated manner. Effective coordination is critical for maintaining model version consistency, facilitating seamless collaboration among contributors, and ensuring that HF users have timely access to the latest improvements and fixes.

However, synchronization between GH and HF repositories is not always straightforward. As illustrated in Figure 2, discrepancies in synchronizing development activities frequently occur across different PTLM families (one or more variant of PTLMs that are managed on a upstream GH repository), where updates fail to propagate in a timely or consistent manner. This highlights an inconsistency across the release times between the two platforms, which can result in outdated or incomplete information on one platform compared to the other. For instance, the last updated PTLM in the YI family (see the red boxes), 01-ai/Yi-34B-Chat[12], received a template update on HF on November 10, 2024, while the corresponding GH repository was not updated for over two weeks. Similarly, in the Magicoder family, the last updated PTLM, ise-uiuc/Magicoder-DS-6.7B[13], had its license and chat template modified on HF on March 6, 2024, whereas a citation update appeared on GH only on November 1, 2024—a coordination gap of nearly eight months. In the Indobart family, the last updated project, indobenchmark/indobart-v2[14], received a system prompt update on HF on June 21, 2022, while a corresponding benchmark version change on GH occurred more than a year later, on December 5, 2023.

These examples highlight the lack of systematic coordination between GH and HF repositories, with updates often appearing on one platform long before—or after—they are reflected on the other. The extent of these delays varies, suggesting an inconsistent flow of development activities across platforms. This study analyzes these patterns by identifying the characteristics influencing coordination and delays, examining the different synchronization patterns that emerge, and assessing how common these patterns are across various PTLM families.

[12] https://huggingface.co/01-ai/Yi-34B-Chat
[13] https://huggingface.co/ise-uiuc/Magicoder-DS-6.7B
[14] https://huggingface.co/indobenchmark/indobart-v2

2.4 Related Work

2.4.1 Commit classification and taxonomy

Understanding software changes through commits has been a long-standing research area, with early efforts focusing on developing taxonomies to categorize changes based on their purpose. Hindle et al. (2008) analyzed large commits in software repositories, finding that architectural changes, often associated with perfective maintenance, were more likely to be large, while small commits typically represented corrective changes. Bhatia et al. (2023) extended this taxonomy by adding two high-level categories—ML-specific Data and Dependency Management—and 16 sub-categories, nine of which are specific to machine learning, such as input data, parameter tuning, and model structure. Similarly, (Janke and Mäder, 2024) studied 1,000 Java projects and identified over 45,000 code change patterns, such as "Add attribute, update methods" and "Add handler and reaction," categorizing them into seven high-level groups. They found that while many patterns were specific to individual projects and contributors, a few common patterns appeared across all contexts. This study revealed how different commit patterns, although project-specific, can offer insights into software evolution and help understand the frequency and nature of maintenance activities.

Building on these taxonomies, researchers began to explore how commits could be automatically classified. Recent years have witnessed significant advancements in commit classification, particularly with the application of machine learning models to automatically categorize software changes. Several studies have employed traditional machine learning techniques for classifying commits based on metadata such as commit messages and author identity. For instance, (Mockus and Votta, 2000) trained a classifier for maintenance activities based on textual change descriptions, finding that perfective changes were prevalent in legacy systems and identifying significant relationships between change type, size, and time required for implementation. Furthermore, (Yan et al., 2016) introduced a discriminative model called Probability Latent Semantic Analysis (DPLSA) for classifying software changes into multiple categories, outperforming baseline methods across five open-source projects with improved accuracy and recall. These early approaches paved the way for more sophisticated methods by relying on commit metadata, but the introduction of deep learning models has further enhanced classification performance.

More recently, these classification and taxonomy approaches have been extended to the domain of pre-trained models. Castaño et al. (2024a) leveraged Gemini-1.5 Flash (Google, 2024) to classify commits on the HF platform, utilizing the taxonomy from Bhatia et al. (2023), identifying Training Infrastructure, Output Data, and Project Metadata as the most frequent commit types. While they focused on internal development patterns within HF, we utilized the same taxonomy but adopted a cross-platform perspective, examining how development activities are coordinated—or fragmented—across both GH and HF. Although originally developed for general ML applications, this taxonomy provides a useful approximation for distinguishing commit types in PTLM projects. Based on this, we classified commit activities to identify change types across platforms. While it may not capture all nuances of PTLM development, it enables a first-order comparison of update patterns across the two ecosystems. We further extended this analysis by applying topic modeling to commits within each change type, allowing us to explore the thematic focus of upstream and downstream activities and examine potential directional relationships between platforms. Furthermore, we identified eight distinct synchronization patterns between GH and HF, offering insights into the efforts practitioners undertake to maintain PTLMs across the two platforms. We also examined how characteristics such as variants of PTLMs and cross-platform authors influence synchronization. While Castano et al. study activity dynamics within HF, our work presents a broader view of distributed PTLM development, emphasizing its fragmented and interdependent nature.

2.4.2 Coordination strategies in software development

Coordination has long been recognized as a critical factor in software engineering, dating back to foundational work on task interdependencies and communication bottlenecks (Malone and Crowston, 1994, Herbsleb et al., 2001). Given the vast scope of coordination research, we refer readers to recent surveys such as (Talukder

et al., 2017) for a comprehensive overview. Here, we focus on the most closely related studies that inform our analysis of coordination in the release of PTLMs across collaborative platforms.

Several studies have examined coordination strategies in agile and co-located software teams. Strode et al. (2012), Strode and Huff (2015) developed a theoretical model of coordination in agile environments, identifying synchronization, structure, and boundary spanning as central coordination components. Their model links these strategies to both implicit and explicit coordination effectiveness. Kanaparan and Strode (2025) further validated this theory through a survey of 340 agile practitioners, showing that these coordination strategies significantly impact coordination outcomes, with customer involvement acting as a moderating factor. These findings provide a robust theoretical foundation for understanding coordination in fast-evolving, collaborative software projects.

Expanding to distributed and global settings, researchers have explored the role of digital tools and asynchronous practices in maintaining coordination. Giuffrida and Dittrich (2015) proposed a framework using communicative genres and coordination mechanisms to study distributed software teams, highlighting the importance of social software (SoSo) in establishing and maintaining team protocols. Stray and Moe (2020) studied Slack and meetings in a longitudinal case of global software engineering and found that while distributed teams benefit from collaboration tools, they still face challenges related to availability and participation. Similarly, (Li and Maedche, 2012) examined how agile coordination strategies are adapted in global software development, using Coordination Theory to explain how agile practices replace rigid plan-driven methods in cross-boundary projects.

At larger scale, inter-team coordination has been a topic of growing interest. Berntzen et al. (2021) investigated a 16-team agile development program and identified four coordination strategies: aligning autonomous teams, maintaining project-wide awareness, managing prioritization, and handling architectural dependencies. Their work extends the discussion of coordination from team-level to program-level structures.

Other researchers have taken broader empirical and technical perspectives. Foundjem and Adams (2021) empirically analyzed release coordination in the OpenStack ecosystem and identified ten key coordination activities. Bock et al. (2022) linked developer communication threads to software features, showing that higher-level coordination views reveal stronger associations between commits and communication than lower-level ones. Krause-Glau et al. (2022) developed a cross-device collaborative visualization system for program comprehension, illustrating how real-time synchronization of user events supports coordination in development tasks.

Coordination outside traditional SE contexts also provides insight into collective action. Magelinski et al. (2022), for instance, proposed a multi-view framework for detecting covert synchronized actions on Twitter, demonstrating how coordination detection can extend beyond code repositories to social behaviors in digital ecosystems.

Finally, (Talukder et al., 2017) conducted a systematic literature review of 50 primary studies on distributed agile coordination published between 2006 and 2016. They categorized the literature into theoretical foundations, tools and techniques, and challenges, and highlighted the fragmented nature of existing research—underscoring the need for integrated coordination frameworks in modern development contexts.

In our study, we explore the synchronization of commit and release activities for PTLMs on GH and HF. Our aim is to understand the efforts practitioners make to synchronize actions across these platforms, and to identify synchronization strategies that ensure consistency, traceability, and reliability in the release and evolution of pre-trained models. Unlike prior work, which focuses on general software development or agile team settings, we study synchronization in the context of PTLM ecosystems. Furthermore, while earlier studies often rely on surveys or interviews, we adopt a repository mining approach. Our focus is specifically on timing synchronization of commit activities across platforms, rather than the full spectrum of coordination activities in traditional development workflows.

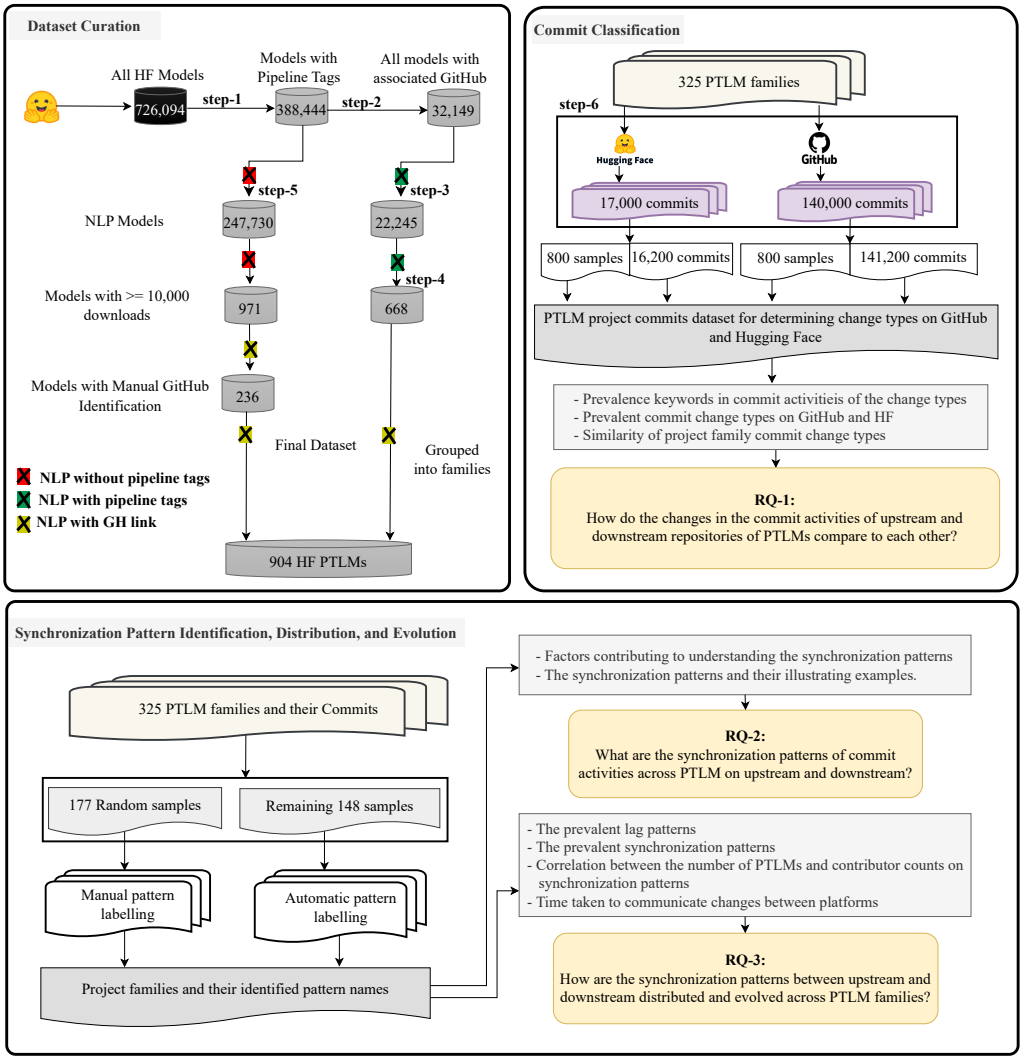

Fig. 3 Data collection procedure

3 Study Setup

This section presents the design of our empirical study. Our overall goal is to investigate the synchronization of commit activities for PTLMs between GH (the upstream platform) and HF (the downstream platform). To explore these synchronization patterns, we examine the three interconnected research questions presented in the introduction. To facilitate our investigation, we first explain our methodology for data collection in Section 3.1, which details the step-by-step process of curating the dataset and forms the foundation for answering the three RQs in Section 4. We illustrate the methodology followed to extract, refine, and analyze the dataset in Figure 3, providing a visual overview of the study's workflow and how it aligns with our research questions.

Algorithm 1 Identifying the Correct GH Link for a PTLM

Input: List of GH links, Model name in the form 'owner/model_name' (e.g. ProsusAI/finbert)
Split the model name into two segments:
    `left_segment` (owner), `right_segment` (model name after '/')
Initialize an empty list `valid_links`
for each GH link in the list **do**
    if GH link contains `left_segment` **then**
        Add the link to `valid_links`
    end if
end for
if `valid_links` is empty **then**
    for each GH link in the list **do**
        if GH link contains `right_segment` **then**
            Add the link to `valid_links`
        end if
    end for
end if
for each link in `valid_links` **do**
    if link does not contain `left_segment` or `right_segment` **then**
        Remove the link from `valid_links`
    end if
end for
Remove duplicates from `valid_links`
Output: List of unique and valid GH links

3.1 Data Collection Methodology

In this study, we selected HF as our downstream platform of choice due to its widespread adoption and central role in PTLM distribution. Unlike other platforms, such as TensorFlow Hub or ONNX Model Zoo, it offers broader model coverage, richer metadata, and stronger community support (Ait et al., 2025). This prominence is reflected in the platform's hosting of over one million models across dozens of tasks. At the same time, we selected GH as our upstream platform, since it serves as the leading open-source platform where core development and collaborative commit activities take place. We outlined the six steps that we followed to obtain our dataset of PTLM commit activities from both HF and GH, ensuring that we accounted for the interaction between the two platforms in our study. Our methodology includes a two-pronged approach: Steps 1 through 4 and Step 6 involve automated data filtering processes, while Step 5 focuses on manual inspection and dataset refinement. The outputs from both streams are then merged to form the final dataset.

- **Step 1: Extracting all models from HF:** We retrieved all models uploaded to HF using the HfApi Client[15], following the extraction procedure shown in Figure 3. As of June 10, 2024, we extracted 726,094 models, with 388,444 (53.5%) categorized into identifiable tag categories defined by HF's tagging system: Audio (27,040), Computer Vision (46,788), Multimodal (853), Natural Language Processing (PTLMs) (269,975), Reinforcement Learning (43,457), and Tabular (331). The remaining 337,650 models (46.5%) do not have identifiable pipeline tags. To ensure consistency, our next filtering step focused on the former models with an identifiable pipeline.
- **Step 2: Automatically identifying GH links for the 388,444 models:** To automatically extract the links to the GH repository from HF model cards, we developed a script (available in our replication package, (Adekunle, 2025)). The challenging part of link identification addressed by this script is that practitioners often include multiple GH links in model cards; however, some of these links may not directly correspond to the model itself—they might reference other models or serve as general references.
  To ensure we identify only the appropriate GH link for each model, we apply several heuristics, as outlined in Algorithm 1. First, we split the model name into two segments using the forward slash (/), since HF structures model names this way to separate the owner segment (on the left) from the model segment (on

[15] `https://huggingface.co/docs/huggingface_hub/package_reference/hf_api`

the right) (Ajibode et al., 2025). Then, we search all GitHub links on the model's card for either the owner or model segment, discarding links that contain neither and removing duplicates. Through this process, we did not identify any model owner with more than one link affiliated with their models. Applying this method to the 388,444 models associated with pipeline tags, we found that 32,149 models (8.27%) have GH links. Since a single GH link may be used by multiple models, we identified 3,702 unique GH repositories. Our dataset includes models from prominent families such as Gemma (Google), Phi-3 (Microsoft), and LLaMA (Meta), which gives us confidence in the representativeness and relevance of the extracted data. We acknowledge that this heuristic approach may exclude some valid repositories if owners use naming conventions that differ substantially from the model name (e.g., project-specific names or acronyms unrelated to the HF identifier). Consequently, certain valid repositories may have been missed, particularly for organizations that maintain many models within centralized repositories using generalized names (e.g., "research-models"). We discuss the impact of this limitation further in Section 6 (Threats to Validity).

- **Step 3: Selecting PTLMs:** Among the 32,149 models with GH links, 22,245 (69%) are NLP models (i.e., text modality models), which span 74% of the 3,702 unique GH repositories. This observation supports the findings of (Castaño et al., 2024b), who reported that NLP models tend to be better documented and more widely adopted. Based on these insights, we focused our subsequent filtering on the NLP models. To support further research, our replication package includes the GH links for both the NLP and non-NLP models.
- **Step 4: Selecting popular PTLMs based on # downloads:** Although 22,245 NLP models had associated GH repositories, we recognize that not all linked repositories are directly related to model development. To assess this, the first author manually reviewed 50 randomly selected models and their GH repositories, examining the presence of training code or replication packages. This review revealed that 22% of the sampled models lacked these resources—and all of them had fewer than 10,000 downloads on HF. In contrast, models with 10,000 or more downloads on HF consistently had well-documented replication packages. For instance, the repository for altsoph/xlmr-AER[16] (27 downloads) contained only PDF files, while the repository for izhx/udever-bloom-560m[17] (5,139 downloads) included only a dataset. In contrast, highly downloaded models like google-bert/bert-base-uncased[18] and wukevin/tcr-bert[19], with 67M+ and 6M+ downloads respectively, provided full training resources.
  To focus on high-quality, well-maintained models, we applied a 10,000-download threshold (Jiang et al., 2023a), reducing our dataset to 668 NLP models linked to 271 GH repositories. This threshold balances the inclusion of widely adopted models (Stalnaker et al., 2025) with the substantial manual effort our study entailed: we verified GitHub links from 971 repositories (≈24 hours), manually labeled 1,600 commits (≈48 hours per author), and reviewed 177 synchronization figures (≈10 hours per author).
  We acknowledge that applying this threshold excludes about 96% of available NLP models, introducing potential sampling bias. These excluded models may represent recent releases, specialized domains, or contributions from smaller research teams. Therefore, our findings are most generalizable to well-established, community-recognized models. We revisit this limitation in Section 6 and encourage future work to assess whether similar trends apply across the broader model ecosystem.
  After applying the threshold, the first author conducted another manual review of 50 randomly selected models to verify the correctness of associated GH links. This step ensured that repository links truly corresponded to the models, particularly in cases where owners (e.g., EleutherAI) manage multiple repositories. For example, EleutherAI's pythia-160m[20] and llemma_7b[21] are maintained in separate repositories. This verification step confirmed the accuracy of the repository-to-model mapping.
- **Step 5 Manually identifying GH links for the top popular PTLMs:** While the previous step focused on the 22,245 PTLM variants with GH links identified in Step-3, here we focused on the remaining 247,730

[16] https://github.com/altsoph/BAER
[17] https://github.com/manueltonneau/twitter-unemployment/tree/main
[18] https://github.com/google-research/bert
[19] https://github.com/wukevin/tcr-bert
[20] https://github.com/EleutherAI/pythia
[21] https://github.com/EleutherAI/math-lm

PTLM variants that were filtered out because they did not have GitHub links in their model cards. To further increase our dataset, we again applied a 10,000-download threshold, prioritizing popular models for manual link identification. This resulted in a subset of 971 high-download PTLMs that we manually analyzed for the presence of GH links. For this manual identification process, we follow the steps outlined below:

- **Exploring the PTLM owner's HF profile to locate a GH link, if available.** If the owner's profile included a GH homepage link, we visited the corresponding profile and reviewed all repositories to identify any containing HF model training resources, such as training scripts, configuration files, or fine-tuning setups.
  For profiles with fewer than 50 repositories, we examined each one individually. For profiles with more than 50 repositories, we conducted a targeted search using the HF model name. If this search returned relevant results, we verified whether the repository actually contained training code and replication packages. Confirming the presence of these resources helped us avoid including repositories created for purposes other than solely managing the HF model.
  For example, YituTech/conv-bert[22] did not specify a GH link in its model card. By visiting the owner's GH profile[23], we discovered the ConvBERT repository[24], which contained relevant training scripts. In contrast, papluca/xlm-roberta-base-language-detection[25] had a GH profile[26], but none of its repositories matched the model name or contained relevant files, so we excluded it.
- **Examining academic publications referenced in the model card:** In a situation where we could not identify the owner's GH repository through their HF profile, we manually examine any academic publication referenced in the model card. If a publication explicitly states that a GH repository was used to store information about the training or related information about the model, we manually inspect the repository to confirm if it has information, such as training script in the GH repository. For instance, the owner of medicalai/ClinicalBERT[27] did not specify a GH link in their model card or profile. However, in their academic publication[28], they state: "The codes are available for academic research and non-commercial use on GH[29]." In such cases, we used publications to identify the corresponding GH repository.

At the end of Step 5, we identified a total of 698 PTLM variants with GH links from 971 well downloaded PTLM models with identified pipeline tags. However, 462 of these repositories did not contain essential training-related resources, which indicated that they were not used for managing HF models. Consequently, we discarded them, leaving 236 models associated with 63 unique GH repositories in addition to the total from Step-4.

By combining both manual and automated methods from steps 4 and 5, we ultimately identified 904 HF PTLMs associated with 325 unique GH links, constituting our final dataset. To further consolidate the dataset, we grouped multiple HF models under the same GH repository into a "family"—a set of PTLMs managed within a single GH repository—capturing the coordinated development and reuse practices commonly observed in such model groups. For example, the following four PTLMs on HF—neulab/codebert-

[22] https://huggingface.co/YituTech/conv-bert-base
[23] https://github.com/yitu-opensource
[24] https://github.com/yitu-opensource/ConvBert
[25] https://huggingface.co/papluca/xlm-roberta-base-language-detection
[26] https://github.com/LucaPapariello
[27] https://huggingface.co/medicalai/ClinicalBERT
[28] https://doi.org/10.1038/s41591-023-02552-9
[29] https://github.com/rlditr23/RL-DITR

python[30], neulab/codebert-cpp[31], neulab/codebert-java[32], and neulab/codebert-c[33]—are all maintained within a single GH repository[34].

– **Step 6 Extracting Commits from 904 HF PTLMs and their respective 325 GH repositories:** To understand what is happening on GH and HF and how their activities are different or similar to each other, we extracted commit activity from both platforms. Commits were chosen as the focus of our analysis because research has established them as fundamental components of software development, and used to study software evolution (Lin et al., 2013), maintenance activities (Heričko and Šumak, 2023), and developer collaboration (Tian et al., 2022). Commits represent finalized work, whereas pull requests and issues often reflect discussions or work in progress. Additionally, HF hosts relatively few pull requests and issues compared to GH, further supporting our decision to concentrate solely on commit data. Commit information was extracted from both HF and GH on October 13, 2024.
To extract commit activities from HF, we developed a script (available in our replication package (Adekunle, 2025)) that uses the HF HfAPI Client to retrieve repository metadata, including commit titles, messages, authors, and timestamps. Before outputting results, we apply preprocessing steps such as Unicode normalization[35], removal of HTML tags and code blocks, and collapsing excessive whitespace. These steps ensure data consistency, improved readability, and the removal of formatting artifacts while preserving essential commit content. This process resulted in 17,000 commits extracted from HF repositories.
Similarly, to extract commit activities from GH, we developed another script (available in our replication package (Adekunle, 2025)) that utilizes the GH REST API[36] to retrieve commit messages, authors, and timestamps. To maintain consistency, we apply the same preprocessing steps as for HF, including the removal of HTML tags, and code snippets while normalizing whitespace. To handle API rate limits, we implement automatic request throttling and retries. This process resulted in 140,000 commits extracted from GH repositories.
The remaining components of our methodology, as outlined in Figure 3, are discussed within the context of each research question in the subsequent sections. Rather than presenting all methodological steps in a single block, we integrate them into the structure of the research questions to maintain clarity and provide targeted explanations that align with the specific goals of each RQ.

4 Results

4.1 $\mathbf{RQ}_1$: How do the changes in the commit activities of upstream and downstream repositories of PTLMs compare to each other?

Downstream PTLM owners reference GH repositories for various purposes, as specified in model cards. These include providing code for pretraining and fine-tuning[37], reporting issues with the model[38], and sharing additional resources[39], such as usage examples and tutorials. However, it remains unclear to what extent PTLM stakeholders—including developers, maintainers, and users such as app developers—ensure that changes made on one platform are communicated to the other. To address this, we investigate the prevalent change topics (subcategories) in the commits of PTLM families on GH and HF, quantify the change types, and calculate the degree of similarity of these change types. By analyzing these aspects, we aim to shed light on the extent

[30] https://huggingface.co/neulab/codebert-python
[31] https://huggingface.co/neulab/codebert-cpp
[32] https://huggingface.co/neulab/codebert-java
[33] https://huggingface.co/neulab/codebert-c
[34] https://github.com/neulab/code-bert-score
[35] https://unicode.org/reports/tr15/
[36] https://docs.github.com/en/rest?apiVersion=2022-11-28
[37] https://huggingface.co/m3rg-iitd/matscibert
[38] https://huggingface.co/meta-llama/Llama-2-70b-hf
[39] https://huggingface.co/openai-community/gpt2-medium

of the upstream-downstream relationship between GH and HF platforms, which is crucial for understanding how PTLMs are managed across platforms. This investigation not only helps us understand the current state of cross-platform integration but also provides valuable insights for researchers and model developers, potentially guiding efforts to streamline the release processes of PTLMs.

4.1.1 Approach

4.1.1.1 Classifying the change types of PTLM families on GitHub (GH) and HuggingFace (HF) using manual and automated methods. To classify PTLM commit change types (changes categorized at the top level of abstraction) in HF and GH, we employed both manual and automated methods. Before beginning this analysis, (1) the first, second, and third authors held a roundtable discussion to collaboratively define the categories for labeling commit activities. This discussion focused on determining the suitability of the categories established in previous research (Castaño et al., 2024a) and (Bhatia et al., 2023). (2) Drawing from the ideas in these two studies, we reviewed the provided examples in (Castaño et al., 2024a)'s study to ensure the categories could be applied consistently to the dataset of PTLMs. (3) After agreeing upon the suitability of the categories and to classify the commits into their respective change type categories, the first author commenced the manual analysis, categorizing 800 commits from each of the HF and GH platforms, totaling 1,600 commits. This manual analysis not only ensured the applicability of the established categories to the PTLMs but also provided a ground truth for evaluating the performance of the automated labeling process using Gemini-1.5 Flash. (4) Following the manual analysis, we used Gemini-1.5 Flash (Google, 2024), a large language model, to automatically label the same 1,600 manually labeled commits, enabling us to assess the feasibility of Gemini 1.5 Flash to categorize commits. (5) After confirming the suitability of Gemini Flash 1.5 for our task by using it to label the already manually labelled commits, and also demonstrated in prior research (Castaño et al., 2024a), we used this LLM to label the remaining 16,200 and 139,000 commits on HF and GH. (6) Finally, we visualized the distribution and prevalence of these commit change types using bar charts. The complete process involves the following steps:

Step 1.1: Selection of representative samples of commits from GH and HF. Given the large volume of commits on both platforms (17,000 from HF and 140,000 from GH), we used a two-stage stratified random sampling approach (Aubry et al., 2023) to ensure representation across diverse commit types while keeping the manual labeling workload manageable. In the first stage, we randomly selected 10 of the 325 families in our dataset (see Step 0.5 in Section 3.1 for how we consolidated PTLMs into families). During preliminary observations of the temporal relationship between GH and HF activities, we identified eight distinct temporal patterns (i.e., differences in the timing and alignment of commit activities across platforms), which we later formalized as synchronization patterns in RQ2. To ensure that all these patterns were represented within the selected families, we randomly sampled 10 commits per family for each pattern, resulting in a total of 800 commits per platform (8 patterns × 10 families × 10 commits). While the small number of families was chosen to ensure feasibility for manual analysis, the predefined taxonomy guided our labeling of these commits. This sample was not intended to develop a new taxonomy, but to validate the applicability of existing categories and create a reliable ground truth for evaluating the performance of our LLM-based classification method.

Although stratified random sampling is often used for statistical generalization, our aim was to proportionally represent all identified patterns of temporal relationship in our preliminary observation. This approach allowed us to capture diverse commit activities while keeping the dataset size manageable for thorough manual labeling.

Step 1.2: Manual and automatic labeling of representative samples of commits. Following the selection of 800 commits from each of GH and HF, the first author manually labeled each commit based on the 15 different commit change types (taxonomy) reported in Castaño et al. (2024a), Bhatia et al. (2023). This taxonomy was specifically developed to classify changes in commit activities of AI models, providing a structured framework for understanding the evolution and refinement of machine learning PTLM families over time. While Castaño et al. (2024a) primarily focused on HF, the taxonomy itself was adapted from Bhatia et al. (2023), which specifically developed and validated taxonomies using GH commits. Therefore, the taxonomy is inherently

applicable to GH as well. Since PTLMs are a specialized subset of machine learning projects, this taxonomy remains relevant for analyzing commit histories on both platforms.

This taxonomy categorizes commit activities into the following 15 types:

- Preprocessing: Changes to data transformation before model training (e.g., tokenizer fixes, normalization).
- Parameter tuning: Adjustments to hardcoded hyperparameters (e.g., learning rate, batch size).
- Model structure: Modifications to the model's architecture or code (e.g., layer changes, bug fixes).
- Training infrastructure: Updates to training logic (e.g., checkpointing, distributed training setup).
- Pipeline performance: Optimizations for runtime efficiency (e.g., faster data loading, memory fixes).
- Sharing: Changes enabling collaboration (e.g., Git hooks, shared configs, CI/CD workflows).
- Validation infrastructure: Modifications to evaluation logic (e.g., new metrics, benchmark updates).
- Internal documentation: Developer-facing documentation (e.g., code comments, merges.txt).
- External documentation: User-facing documentation (e.g., README.md, model cards).
- Input data: Changes to data loading/ingestion (e.g., new datasets, correct a path to datasets).
- Output data: Adjustments to output storage (e.g., saving predictions in a new format).
- Project metadata: Non-functional updates (e.g., versioning, licensing, initial commits).
- Add dependency: Introduction of a new library/package (e.g., adding transformers).
- Remove dependency: Removal of a library/package (e.g., dropping flax).
- Update dependency: Updates in version/metadata (e.g. transformers$>= 4.29$).

After the first author completed the manual labeling of 800 commits on each platform, we developed a Python script (available in our replication package (Adekunle, 2025)) that leveraged a large language model to label the remaining 16,200 commits on HF and 141,200 commits on GH. Specifically, we embed Gemini-1.5 Flash (Google, 2024) in our script to perform automatic annotation, using the 15 taxonomies and their descriptions as guidance. To improve the accuracy of commit labeling by the LLM, we provided it with examples of our manually labeled commits that represent each taxonomy. An example of the prompt provided for LLM is given in Appendix A. We selected Gemini-1.5 Flash due to its balance of efficiency, cost-effectiveness, and speed in classifying commit messages and other textual data. It provides performance comparable to GPT-4 while being 50 times cheaper and significantly faster—a choice further validated by its successful use in prior commit classification work (Castaño et al., 2024a). To maintain reproducibility, we used Gemini-1.5 Flash in its default API configuration. Given that we manually labeled a substantial portion of the dataset, the model served to scale up the annotation process efficiently, as LLMs like Gemini-1.5 Flash are known for their rapid text classification capabilities (Castaño et al., 2024a) compared to manual labeling.

For GH commits, we provided the raw commit messages directly to the model. In contrast, for HF commits, we concatenated the commit titles and messages into a single entry. This was done to accommodate cases where commit messages were empty and to ensure that the LLM had sufficient context for accurate classification. At the time of data preparation, we treated GH commit messages as standalone entries. Although the GH API provides a single commit message string, it is a standard convention that the first line serves as the title and the subsequent lines form the body. In contrast, the HF API distinguishes titles and messages explicitly.

To assess the reliability of the LLM-based labeling process, we repeated the automatic classification 10 independent times for each platform. Each run produced a complete labeling of all commits. Rather than aggregating labels from these runs, our goal was to evaluate the consistency of the LLM's output across runs. This method aligns with hallucination-detection techniques such as SelfCheckGPT (Manakul et al., 2023), which analyze variability across multiple LLM responses (e.g., $y_1, y_2, \ldots, y_n$) to assess output stability. We did not aggregate the 10 labels into a final decision per commit, since our objective was not to finalize commit labels but to evaluate the overall labeling performance. To quantify this, we compared each of the 10 automatically labeled sets against the manually labeled subset of 800 commits using Cohen's Kappa (Vieira et al., 2010). The average agreement across these 10 comparisons was $\kappa = 0.72$ for HF and $\kappa = 0.79$ for GH, indicating substantial agreement (Pérez et al., 2020).

Step 1.3: Automatic labeling of the remaining commit messages. After assessing the agreement between Gemini-1.5 Flash and human labeling using Cohen's Kappa score, we applied the LLM to categorize the remaining 16,200 commits on HF and 141,200 commits on GH from the 315 PTLM families. To ensure consistency in categorization across the entire dataset, we used the same 15 taxonomy categories.

4.1.1.2 Quantifying the distribution of PTLM family change types on GH and HF. To quantify the distribution of change types in PTLM PTLM family commits on GH and HF, we wrote a Python script (available in our replication package (Adekunle, 2025)) to aggregate the occurrences of each labeled activity type and calculate their relative proportions across both platforms. During processing, we found that 41 out of 140,000 GH commits were assigned labels not included in our predefined taxonomy of 15 change types—specifically, "bug fix" and "add model." Upon manual inspection, we determined that these labels were misclassifications: all 41 commits aligned with the "model structure" category in the taxonomy and are manually fixed. The first author corrected these assignments accordingly. We then grouped the validated commit change types by platform and computed their proportions to assess differences in prevalence. This analysis revealed notable differences in how practitioners engage with PTLM families on GH and HF during model development.

4.1.1.3 Automatic identification of prevalent topics in each change type of commit activities on GH and HF. To gain a clearer understanding of the dominant development areas within each change type, we applied topic modeling techniques that reveal granular insights into the actual changes occurring on GH and HF—going beyond the broader similarities and differences captured by our change type taxonomy.

To identify change topics within different change types, we developed a script—available in our replication package (Adekunle, 2025)—that applies topic modeling using BERTopic[40]. This method enables a fine-grained classification by grouping semantically similar topics within each change type. BERTopic leverages transformer-based embeddings to capture contextual relationships between words—allowing it to group terms like "optimize," "enhance," and "improve" based on semantic similarity rather than simple co-occurrence. This gives it a significant advantage over traditional topic modeling methods like bag-of-words and Latent Dirichlet Allocation (LDA), which rely on word frequency. BERTopic has also demonstrated strong performance in empirical software engineering studies, where it has been used for categorizing software maintenance activities and analyzing documentation (Grootendorst, 2022, Diamantopoulos et al., 2023, Gu et al., 2023, Zhao et al., 2024, Chagnon et al., 2024).

To prepare the dataset for BERTopic, we categorized commit messages by change types for both GH and HF repositories. For the commit message of each change type, we removed URLs, platform-specific stopwords, short terms (fewer than three characters), and file extensions. We also excluded generic or non-contributory words such as "com" or repeated references to "GH" or "HF". Finally, we input the cleaned commit messages into the BERTopic model to generate clusters of related terms.

Since the goal was to identify a wide range of topics without imposing prior constraints, we allowed BERTopic to dynamically determine the number of topics based on the intrinsic structure of the data. Instead of pre-specifying k, we employed HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise) to determine topic clusters, which allowed the model to adapt to the data's natural clustering tendencies. This approach helps mitigate the risk of overfitting (by not forcing topics that are too granular) or underfitting (by not overlooking nuanced semantic clusters). To ensure that the number of topics was optimal, we also inspected the topic coherence scores to validate the model's effectiveness in identifying meaningful clusters. To ensure meaningful results, we filtered out low-value or non-informative words, such as "joao," "delete," and "init," after the topics had been generated.

For the topics in each change type, we ranked the topics by the significance of their terms, as determined by BERTopic's internal scoring mechanism. This allowed us to identify the most significant and representative topics within each category. The internal mechanism of BERTopic prioritizes terms that are both frequent within a specific change type and discriminative across the entire set of commit messages. This ensures that we focus on terms that capture the unique aspects of each change type, rather than common, non-informative terms. This ranking process was applied consistently for both GH and HF repositories. In cases where there were fewer than 10 topics or none at all due to insufficient data on HF, we used "No identifiable topic" as a placeholder to maintain consistency in the study.

4.1.1.4 Determining the maturity age group for PTLM family. We calculated the age of each PTLM by subtracting the creation date of each PTLM within a family from the dataset extraction date on HF (June 10, 2024). To estimate the maturity of each PTLM family, we computed the average age of all PTLMs in the

[40]https://github.com/MaartenGr/BERTopic

family. For instance, if family A contains three PTLMs aged 200, 100, and 30 days respectively, the average age is 110 days. We considered—but avoided—using the oldest PTLM's age as the sole proxy for family maturity. While this would highlight early model releases, it could exaggerate the maturity of families that have one old version but several much newer additions. Conversely, using the average age mitigates the impact of such outliers, offering a more balanced view of the family's development timeline. We acknowledge, however, that averaging also simplifies temporal variation within families. Despite this limitation, we believe it offers a reasonable approximation of overall model maturity—especially in cases where active development leads to multiple variants with varying release dates.

Based on this approach, we categorized each PTLM family into one of three groups—Recent, Intermediate, and Matured—using the average age of the PTLMs within each PTLM family. Specifically, we applied quantile-based binning to divide the PTLM family into these groups, ensuring roughly equal-sized categories. Quantile binning was chosen because the distribution of model ages was highly skewed, with many younger projects and a few older ones; using equal-sized bins based on quantiles avoids imbalanced groupings and enables meaningful comparisons across maturity levels. The thresholds at 113 and 430 days reflect intuitive stages in the life cycle of PTLMs: projects under 113 days are likely still under initial development or early adoption, those between 114 and 430 days are in a growth phase, and those older than 430 days tend to be more stable or mature. The Recent, Intermediate, and Matured groups contain 109, 108, and 109 PTLM families, respectively, with minimum average ages of 1, 116, and 436 days and maximum ages of 113, 430, and 1867 days.

4.1.1.5 Examining the dominant commit change types of PTLM family across age group. To determine whether dominant commit change types persist across PTLM families of varying maturity, we analyzed the distribution of change types on GH and HF, stratified by PTLM family age group. Age-group classification methods are detailed in Section 4.1.1.4. We visualized the relative frequencies of change types within each age group using symmetric bar plots. This reveals whether dominant commit change types are consistent or shift with PTLM family maturity.

To test for significant differences in change type distributions between platforms, we applied chi-square tests of independence (McHugh, 2013). For each age group (recent, intermediate, mature), we constructed a 15×2 contingency table, where each row represents one of the 15 predefined change types and each column corresponds to a platform (GH or HF). The cell values contain the frequency of each change type on each platform. The null hypothesis (H_0) assumes that the distribution of change types is the same across both platforms, while the alternative hypothesis (H_1) posits that the distributions differ significantly. The chi-square test is appropriate here because it assesses associations between nominal variables based on frequency counts, without requiring assumptions about normality or continuous measurement.

These tests were conducted separately for recent, intermediate, and mature projects to assess whether platform-specific change types vary by age. After conducting the chi-square tests, we applied a Bonferroni correction to adjust for multiple comparisons, ensuring that the PTLM family-wise error rate was controlled. Specifically, we adjusted the significance level (α) by dividing it by the number of tests conducted, which in our case was three (one comparison per age group). This correction was applied to each p-value, and the results were interpreted accordingly to mitigate the risk of Type I errors.

Additionally, to quantify the strength of the observed relationships, we calculated Cramér's V effect size (Akoglu, 2018) for each PTLM age group. Cramér's V was used because it provides a measure of association strength for categorical variables, helping us understand the magnitude of the relationship between platform and activity type. We interpreted Cramér's V following the guidelines provided in (Akoglu, 2018). According to these guidelines, values greater than 0.25 indicate a very strong relationship, values greater than 0.15 indicate a strong relationship, values greater than 0.10 indicate a moderate relationship, values greater than 0.05 indicate a weak relationship, and values of 0 or very weak indicate no relationship. These effect size measures provide practical insight into the degree to which platform-specific differences influence activity distributions in PTLM development, helping to contextualize the statistical significance.

4.1.1.6 Calculating the similarity score of change types between GH and HF. To calculate the similarity of activities between GH and HF during the PTLM development process, we computed the similarity of activities

across both platforms based on their occurrence within PTLM families. We used the Jaccard similarity coefficient, which measures the proportion of shared elements between two sets relative to their union (Ivchenko and Honov, 1998). Specifically, it quantifies the extent of overlap in the types of commit change types on both platforms, helping us assess whether similar activities are performed across GH and HF for the same PTLM. The resulting Jaccard similarity scores range from 0 to 1, where 0 indicates no commonality in commit change types, and 1 indicates complete similarity. High similarity scores (0.5+) suggest that the same activities are occurring on both platforms, indicating that GH and HF are involved in similar tasks related to the development and maintenance of models. Moderate similarity scores may suggest that the platforms share some activities but also engage in distinct ones, while low scores would indicate that the activities on the platforms are largely independent.

To visualize the distribution of similarity scores, we created a histogram where the y-axis represents the percentage of PTLM families, and the x-axis represents the similarity scores. This visualization provides insight into how frequently the same activities occur across both platforms, highlighting whether there is substantial overlap or divergence in the types of changes recorded.

4.1.1.7 Determining whether maturity and contributor overlap are associated with similarity scores. Beyond computing similarity scores of commit activities between GH and HF, we investigated how these scores relate to model maturity and contributor overlap to better understand their influence on cross-platform synchronization. We first categorized the similarity scores into three groups—low, moderate, and high—using tercile-based quantile binning (qcut[41]) to ensure balanced distribution across PTLM families. The resulting bins were: low similarity (0.00–0.33), moderate similarity (0.35–0.46), and high similarity (0.50–1.00). We then analyzed these similarity categories within model family age classifications (Recent, Intermediate, Matured) to identify trends in activity similarity across different stages of model development.

To assess contributor overlap across platforms, we identified GH profile links of contributors associated with both GH and HF repositories. Since many PTLM families had only one shared contributor on both platforms, we adopted a simple classification: PTLM families with exactly one shared contributor were labeled Single-author, while those with more than one were categorized as Multiple-authors. Next, we analyzed the distribution of model family age groups across similarity score categories and visualized these trends using grouped bar charts. We also employed grouped bar charts to examine how contributor overlap influences commit similarity, distinguishing between single-author and multiple-author PTLM families using different shades.

To formally assess the association between contributor overlap and similarity score categories, we performed a Chi-Square test of independence. The null hypothesis (H_0) states that there is no association between contributor overlap (single vs. multiple authors) and similarity score categories (low, moderate, high); in other words, the distribution of contributor types is independent of similarity scores. The alternative hypothesis (H_1) posits that there is a statistically significant association between the two variables.

This combined methodology allowed us to systematically evaluate how model maturity and contributor overlap impact activity similarity while ensuring statistical rigor and mitigating biases caused by skewed distributions.

4.1.2 Results.

GH (upstream) changes focus most commonly on model structure (29.7%), external documentation (21.0%), and training infrastructure (9.0%), while HF (downstream) changes focus most on external documentation (38.8%), preprocessing (16.6%), and model structure (14.4%).

Even though similar types of changes occur across both platforms, their frequency varies in ways that reflect the differing roles of GH (upstream) and HF (downstream) ecosystems. External documentation changes are nearly twice as frequent on HF compared to GH, aligning with the former's focus on distributing pre-trained models that require detailed documentation for users. In contrast, model structure changes are twice as common on GH (29%) as on HF (14%), reflecting the upstream emphasis on active development, where maintainers regularly revise training scripts, architectures, and configuration files. Similarly, preprocessing changes occur

[41] https://pandas.pydata.org/docs/reference/api/pandas.qcut.html

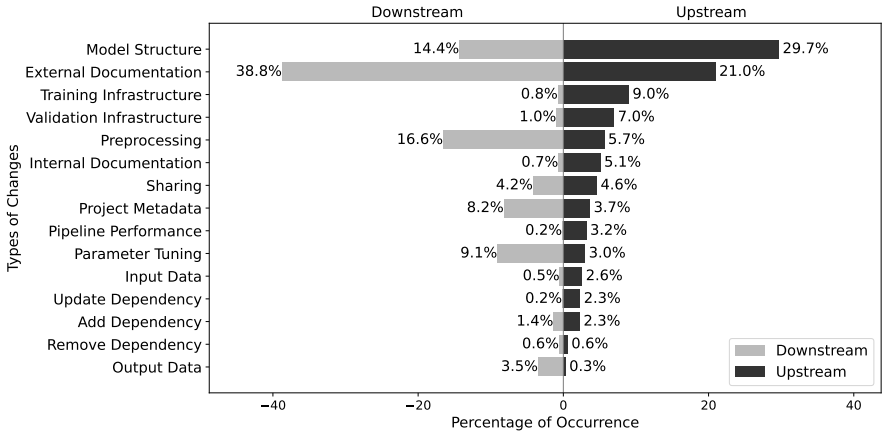

Fig. 4 Proportion of prevalent commit change types in PTLMs on GH and HF.

more frequently on HF (16.6%) compared to GH (5.7%), which could suggest that preprocessing steps are more relevant when models are being prepared for deployment or fine-tuning by downstream users.

Additionally, training infrastructure changes are more prevalent on GH (9.0%) than on HF (0.8%), as GH hosts the code, including training scripts that undergo frequent updates, whereas HF primarily distributes already trained models that require fewer infrastructure changes. These patterns underscore the distinct roles of the platforms in the model development pipeline.

HF focuses on artifact-specific changes, while GH addresses codebase and infrastructure updates. While our earlier analysis presented a high-level categorization of change types across both platforms, we now examine the specific topics that occur within these categories. For instance, although model structure changes are most prevalent on GH, they frequently involve large-scale refactoring, task-specific modifications, and architecture-related revisions. In contrast, HF shows a higher prevalence of external documentation changes, often centered on licensing, prompt templates, and publication citations. This topic-level analysis reveals a diverse set of concerns reflected in the commit activities of PTLMs across both platforms. These topics are discussed in the following section, without implying any specific order of importance or frequency.

Change Types	HF Change Topics	GH Change Topics
Model Structure	**Topics:** Checkpoint, Flax, Architecture, Samplefinetunepy, Mistralforcausallm, Huggingfacehub, Weights, Onnx **Explanation:** The topics characterizing model structure changes primarily involve uploading model weights, defining architecture-specific components, and preparing scripts for model instantiation and integration within the Hugging Face repository.	**Topics:** Bigrefactor, Greedyuntil, Directory, Alexnet, Multiplechoicetask, Makefile, Finetune, Multiheadattention, Remotetracking, Indentation **Explanation:** These topics are more focused on codebase changes, including large-scale refactoring (Bigrefactor), task-specific adjustments (Multiplechoicetask), and structural changes to specific components of the model training scripts (e.g., multi-head attention, finetuning), along with the management of the codebase itself (Makefile, Directory).

External Documentation	**Topics:** Citation, Paper, Latency, License, Template, Docs, Typo, Terms, Create, Transformers **Explanation:** These topics focus on formal and legal documentation, such as research citations, licensing terms, error corrections, and prompt templates for fine-tuning.	**Topics:** Avatar, Custom-moe, Spaces, Acknowledgement, News, Leaderboard, Textdiffuser, Chapter, Header, Featdoc **Explanation:** GH topics are more community-driven, focusing on structural, visual, and feature documentation, as well as tracking contributions and updates.
Training Infrastructure	**Topics:** Enable, Training **Explanation:** These topics reflect efforts to activate or configure training processes—such as initiating fine-tuning runs or modifying training-related settings—often leveraging scripts or model artifacts from GH repositories.	**Topics:** Watchdog, Jupiter, Finetune, Lock, Bump, Setup, Lora, Proxy **Explanation:** GH topics deal with various tools and frameworks for setting up the basic environment—including model training utilities, version management, and task-specific components like fine-tuning—while downstream deploys these setups.
Validation Infrastructure	**Topics:** Inference, Model, Evaluation **Explanation:** These topics focus on the process of running models for inference and evaluation, providing insights into their actual performance through metrics, benchmarks, and leaderboard placements, rather than guaranteeing specific outcomes.	**Topics:** Edit, Wandblogmodel, Baselines, Vision, Squad, Roberta, Leaderboard, Coverage, Monitorpy, Travis **Explanation:** GH topics are more focused on the tools, frameworks, and pipelines used to validate models, including managing evaluation tasks, setting up benchmarks (e.g., Baselines, Squad), monitoring performance (e.g., Coverage, Monitorpy), and ensuring proper test execution (e.g., Travis).
Preprocessing	**Topics:** Fast, Vocab, Tokenizationphismallpy, Language, Files, Tokenizer, Tokenizerconfigjson, Chat **Explanation:** These topics involve preprocessing tasks like tokenization, vocabulary setup, and preparing tokenizer files (e.g., tokenizer_config.json) for language and chat-based models.	**Topics:** Clippy, Localdocs, Quotes, Tatoeba, Glue, alignment, Lint, Cleanup, Wiki **Explanation:** GH topics in this category emphasize tasks like removing noise (e.g., Cleanup, Quotes), ensuring syntactic correctness (Lint, Clippy), and aligning multilingual or task-specific datasets (alignment, Tatoeba, Glue, Wiki).
Sharing	**Topics:** Main, Sync, Huggingfacehub, Safetensors, Duplicate, Model **Explanation:** Focuses on sharing models, managing repositories, handling safe tensor formats, and preventing duplication.	**Topics:** Develop, Xlnet, Commit, Resolve, Gitbook, Origin-big-refactor, Upstreammaster, Bigrefactor, Patch **Explanation:** Emphasizes repository synchronization, major refactoring, and documentation updates.

Internal Documentation	**Topics:** Update **Explanation:** General updates related to internal documentation, possibly covering model descriptions, usage guidelines, or minor adjustments.	**Topics:** Tidy, Length, Isort, Changelog, Flake, Cleanup, Branch, Ruff, Gitignore, Woops **Explanation:** Involves documentation-related efforts for maintaining code cleanliness, formatting, changelogs, and minor refinements.
Pipeline Performance	**Topics:** No identified topics **Explanation:** No specific topics were found related to direct pipeline performance improvements on HF.	**Topics:** Prefetch, Simplify, Feattoolkit, Path, Linted, Optimization, Proxy, Tqdm, Sort, Cuda **Explanation:** These topics focus on optimizing training time, including prefetching data, simplifying code, improving toolkit features, optimizing CUDA operations, and enhancing overall performance.
Parameter Tuning	**Topics:** Modelmaxlength, Config, Configjson, Configjson **Explanation:** These topics focus on modifying model configuration files and setting constraints like maximum model length to optimize hyperparameter settings.	**Topics:** Mcli, Tweak, Global, Criteria, Timeout, Usedevoption, Interval, Anneal, Configjson, Clipping **Explanation:** These topics involve adjusting model behavior through hyperparameter refinements—such as criteria, timeouts, annealing schedules, clipping thresholds, and interval settings—implemented both in code and through configuration files (e.g., configjson), particularly in downstream repositories.
Input data	**Topics:** Dataset **Explanation:** Focuses on modifications to dataset handling, including loading and managing datasets for model training and inference.	**Topics:** Testdata, Samples, Customtraintxt, Tatoeba, Prompt, Arabic, Partial, Json, Template, Branch **Explanation:** Covers a broader range of data-related changes, including test data updates, sample modifications, custom training data, prompt handling, language-specific adjustments (e.g., Arabic), and template structures for input data formatting.
Update dependency	**Topics:** Update **Explanation:** This topic generally refers to broad updates, which may include changes to libraries (and their versions), frameworks, or other components required for running or training the model, though the specific dependencies involved are not explicitly stated in the commit messages.	**Topics:** Pandas, Legion, Sklearn, Flair, Chrome, Scann, Vulnerabilities, Tqdm, Deno, Corenlp **Explanation:** Clearly focuses on updating specific dependencies and libraries, addressing security vulnerabilities, improving compatibility, and enhancing performance by upgrading tools such as Pandas, Scikit-learn (Sklearn), and CoreNLP.

Add Dependency	**Topics:** No identified topics **Explanation:** There are no clear topics indicating the addition of new dependencies, suggesting that dependency additions may not be explicitly highlighted in the commit messages or are handled differently.	**Topics:** Submodule, Initpy, Spacy, Vllm, Pyprojecttoml, Posthog, Reqs, Cmake, Flake, Isort **Explanation:** These topics reflect the explicit addition of new dependencies, including machine learning libraries (Spacy, Vllm), dependency management files (Pyproject.toml, Reqs), and tools for code quality and build management (Flake, Isort, CMake).
Remove Dependency	**Topics:** No identified topics **Explanation:** HF does not show specific topics related to the removal of dependencies. However, this could involve indirect changes to model configurations or related files that no longer use certain dependencies	**Topics:** Directory, Submodule, Transformers, Scripts, Import, Dataset, File, Yaml, Merge, Deepspeed **Explanation:** These topics indicate possible changes in removing dependencies within the GH project. Changes in Directory, Submodule, and Import suggest reorganization or removal of related code, while Scripts, File, Dataset, and Yaml might indicate the removal of code or data files linked to now-removed dependencies. Deepspeed could indicate the removal of an optimization library, and Merge suggests combining code that excludes the removed dependencies.
Project Metadata	**Topics:** Release, License, Commit, Llmfoundry, Basemodel, Link **Explanation:** These topics indicate changes related to project metadata, including licensing updates (License), version releases (Release), foundational model information (Basemodel, Llmfoundry), and commit tracking (Commit). Updates may also include documentation links (Link) for reference.	**Topics:** Codeowners, Create, Unstable, Byml, Dhuang, Chore, Repo, Notes, Multilingual, Adam **Explanation:** These topics relate to project metadata updates, including ownership files (e.g., Codeowners), versioning stability (Unstable), documentation notes, contributor mentions (e.g., Dhuang, Adam), multilingual support, workflow refinements (Chore), and metadata structuring (Byml).
Output Data	**Topic:** Modelsafetensorsindexjson, Pytorchmodelbin, Onnx, Files, Safetensors, Huggingfacehub **Explanation:** Topics focus on how model files are saved, structured, or converted across different formats—such as PyTorch, ONNX, or Safetensors—and how these formats are managed or uploaded to repositories like the HF Hub.	**Topic:** Outputs, Piperoriginrevid, Json, Output, Merge, Meta **Explanation:** Topics indicate changes related to output data management, including handling JSON outputs, managing versions, and merging output files.

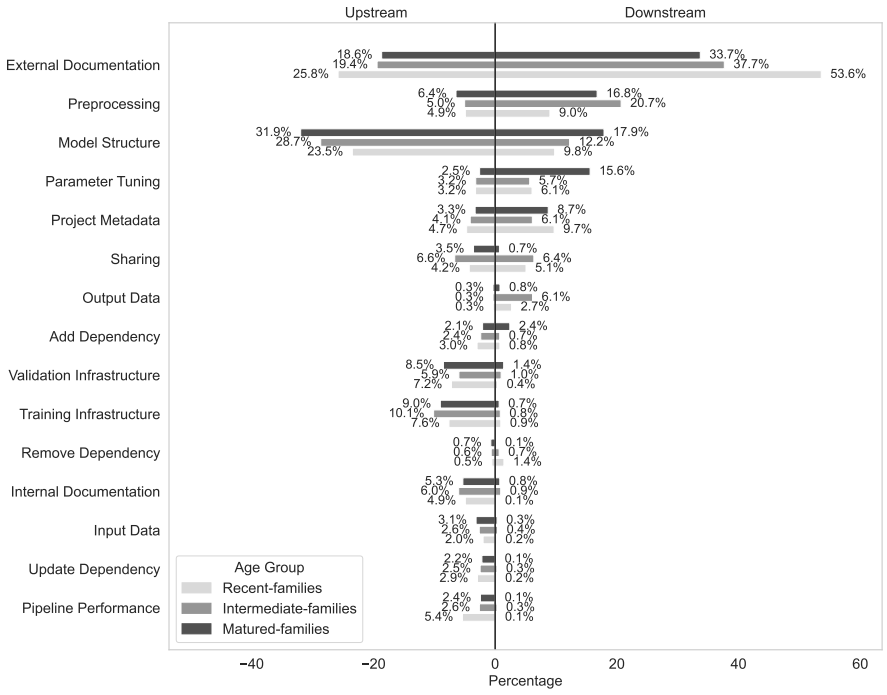

Fig. 5 Variation in the distribution of prevalent PTLM change types across model maturity stages on GH and HF, highlighting shifting emphases on external documentation, model structure, preprocessing, and training infrastructure.

Summary

On the downstream HF platform, updates primarily involve external documentation—such as licensing, prompt templates, and publication citations—while upstream changes on GH focus exclusively on the codebase, including large-scale refactoring, task-specific modifications, and architecture revisions, with limited overlap in the changes across the two platforms.

As models mature, the proportion of prevalent commit change types evolves, with external documentation decreasing, model structure increasing, and training infrastructure and preprocessing showing differing trends across GH and HF. The results in Figure 5 reveal distinct patterns in how the prevalence of external documentation, model structure, preprocessing, and training infrastructure changes over time. While some change types exhibit a consistent trend across family maturity groups, others show varying patterns depending on the commit type, with external documentation decreasing in prevalence over time, and model structure changes becoming more prominent as the models mature.

For recent models, we observe a different trend. On GH, “external documentation” (25.8%) ranks first, followed by “model structure” (23.5%), while “training infrastructure” (7.6%) has a lower proportion compared to intermediate models. On HF, “external documentation” is even more dominant (53.6%), while “model structure” (9.8%) and “project metadata” (9.7%) emerge as notable activities. The increased prominence of “project metadata” — referring to non-functional updates such as versioning, licensing, or initial setup — suggests a downstream emphasis on curating and organizing repositories for public release. For recent models, this likely reflects the effort to establish foundational information that improves discoverability, compliance, and reusability on the HuggingFace hub.

For intermediate models, the rankings of commit change types reflect partially overlapping priorities across both platforms. On GH, “model structure” is the most prevalent activity (28.7%), followed by “external docu-

mentation" (19.4%) and "training infrastructure" (10.1%). On HF, "external documentation" remains dominant (37.7%), followed by "preprocessing" (20.7%) and "model structure" (12.2%). These distributions suggest that intermediate models represent a phase where upstream developers continue refining the internal architecture and training setup, while downstream maintainers focus on improving accessibility through documentation and input adaptations. The recurrence of "model structure" and "external documentation" across both platforms points to their shared importance in bridging core development and user-facing deployment.

For matured models, the distribution shifts further. On GH, "model structure" (31.9%) becomes the most prevalent activity, surpassing "external documentation" (18.6%), while "training infrastructure" remains stable at 9.0%. On HF, "external documentation" continues to decline (33.7%), while "model structure" (17.9%) and "preprocessing" (16.8%) remain significant. Notably, preprocessing decreases in matured models compared to intermediate ones, possibly reflecting that major data-related adjustments (e.g., tokenization, formatting) are typically finalized during earlier stages. As models reach maturity, the focus may shift from data curation to refining architecture and improving user-facing documentation, especially for deployment and reuse.

General trends indicate that GH consistently features the same top activities across all PTLM family ages, though their relative importance shifts. "External documentation" decreases as models mature, while "model structure" increases. "Training infrastructure" peaks at the intermediate stage before slightly declining in matured models. On HF, "external documentation" also declines as models mature, whereas "model structure" consistently increases. "Preprocessing" is more prominent in intermediate models than in matured ones, and "project metadata" emerges as a key focus in recent models.

These shifts reflect how development priorities evolve throughout the model lifecycle. As models mature, focus transitions from public-facing readiness (e.g., documentation, metadata) to internal improvements like structural refinements. The rise in "model structure" suggests ongoing architectural optimization, while the decline in "preprocessing" implies that major data-handling decisions are typically resolved earlier. The prominence of "project metadata" in recent models aligns with the initial setup of repositories for public release — especially downstream — where licensing, versioning, and discoverability are key. Together, these trends underscore the complementary roles of GH and HF efforts, with GH emphasizing core engineering and HuggingFace focusing on accessibility and reuse.

The chi-square tests reveal statistically significant differences in activity distributions between GH and HF at all maturity stages. For recent models, the chi-square value is 2675.86 ($p < 0.0000$, df = 14) with Cramer's V of 0.1961, indicating a strong effect size. Intermediate models yield a chi-square value of 7391.76 ($p < 0.0000$, df = 14) and Cramer's V of 0.2698, suggesting a very strong effect size. For matured models, the chi-square value is 5981.77 ($p < 0.0000$, df = 14) and Cramer's V of 0.2164, again indicating a strong effect. All results below the Bonferroni-corrected threshold ($p < 0.0167$), confirming statistically significant differences between platforms.

Summary

GH shows more changes to model structure, external documentation, and training infrastructure, while HF emphasizes external documentation, preprocessing, and project metadata—changes that are statistically distinct and shift with model maturity.

PTLM families exhibit moderate to high similarity scores in their commit change types across GH and HF, with varying degrees of overlap, as indicated by the analysis in Figure 6. The distribution of Jaccard similarity scores in PTLMs indicates that while 77% of PTLM families (PTLM families with a similarity score of more than 0.3) show similar types of changes between the two platforms, others demonstrate distinct but overlapping activities. A small percentage of families (0.3%) show minimal overlap (0.0-0.1), suggesting a significant divergence in the roles the two platforms play for these families. A larger group (3.7%) falls within the 0.1-0.2 range, where some overlap is observed, but the activities on both platforms still differ considerably. A significant portion of families (18.5%) show moderate overlap (0.2-0.3), indicating that GH and HF share many common commit change types, but with notable differences in how they are used.

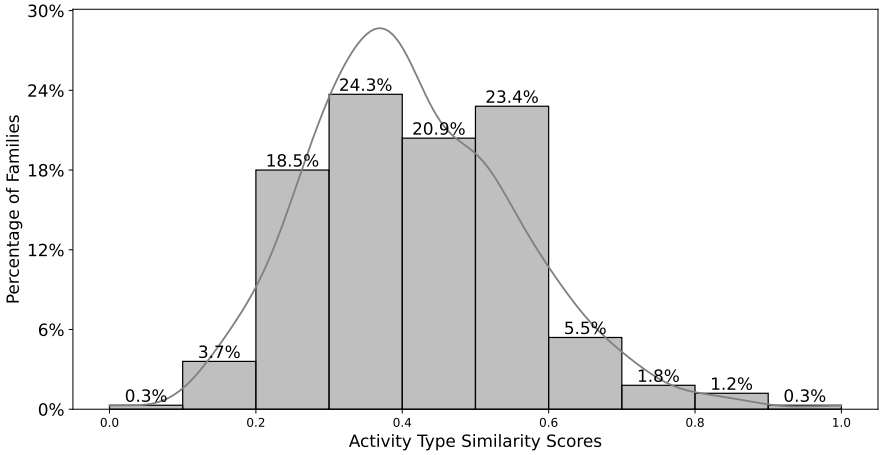

Fig. 6 Jaccard similarity distribution of commit change types across PTLMs, showing varying degrees of activity synchronization between GH and HF.

The largest group of families (24.3%) falls within the 0.3-0.4 range, highlighting that for many families, the platforms share a significant portion of commit change types, though they are not fully aligned. Another substantial proportion (20.9%) of families shows moderate to high similarity (0.4-0.5), reflecting that while both platforms serve similar functions, some divergence remains. Over 20% of families (23.4%) exhibit a high level of overlap (0.5-0.6), suggesting that these families use both platforms in similar ways, although not entirely coordinated.

Smaller proportions of families show even higher similarity. About 5.5% of families fall within the 0.6-0.7 range, and 1.8% fall within the 0.7-0.8 range, indicating increasingly similar roles between the platforms. Only 1.2% of families show near-perfect overlap (0.8-0.9), and a very small fraction (0.3%) exhibits almost complete similarity (0.9-1.0), suggesting that these families use GH and HF in nearly identical ways.

Overall, the results suggest that while GH and HF often serve complementary roles in the model development process, their usage patterns vary across PTLM families.

Model maturity correlates with the similarity score of change types between GH and HF, with the highest alignment observed in the intermediate stage. To explain the previously observed differences in Jaccard similarity scores across platforms, we hypothesize that model family maturity may shape the alignment of activity types. Analyzing the distribution of PTLM families across similarity score categories by family age (Figure 7) reveals a non-linear relationship. Among recent families, a majority (52.7%) fall into the low similarity category, indicating substantial divergence in platform activities for newer models. As models progress to the intermediate stage, the proportion in the low similarity group decreases to 29.1%, while more families fall into the moderate (32.7%) and high (38.2%) similarity categories—suggesting increasing synchronization of activities between platforms.

For matured families, the largest proportion (41.0%) falls into the moderate similarity score category, while the share of families in the high similarity score category decreases slightly to 34.3% compared to intermediate families. The proportion of families in the low similarity score category declines further to 24.8%, showing that fewer PTLM families exhibit strong divergence in platform activities. However, the decrease in the proportion of families with high similarity scores suggests that full convergence between GH and HF does not occur as models mature.

These results indicate a non-linear association between model maturity and the similarity score of changes between GitHub and Hugging Face, with the highest synchronization occurring at the intermediate families. This period often reflects a phase of active refinement for broader release, prompting simultaneous updates to core components, documentation, and usability features across both platforms. As models mature further,

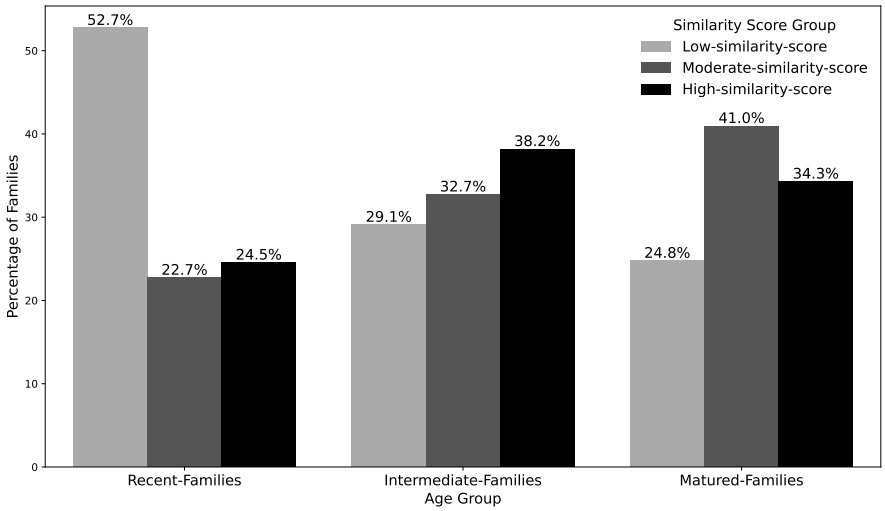

Fig. 7 Similarity score (the higher the more similar) distribution by PTLM family maturity stage, highlighting peak synchronization between GH and HF activities in intermediate models.

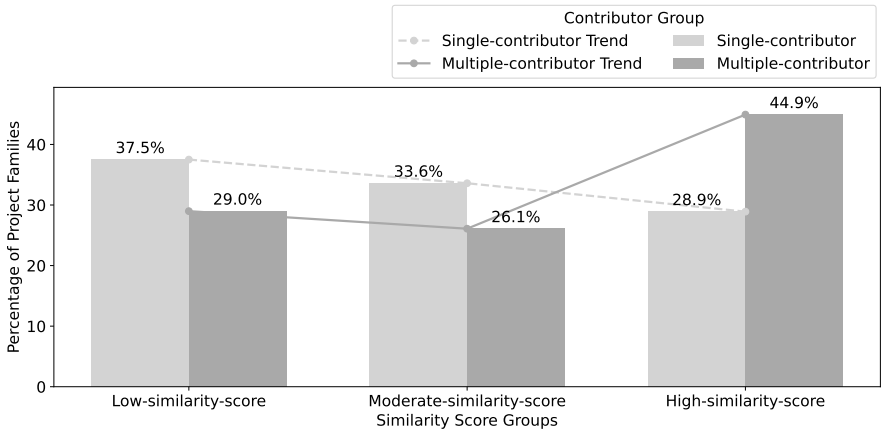

Fig. 8 Cross-platform contributor distribution across similarity score categories, showing that higher similarity between GH and HF activities is associated with multiple shared contributors.

upstream work tends to stabilize while downstream repositories shift toward maintenance, usage facilitation, or curation—resulting in a greater share of moderate similarity scores.

There is also an association between the similarity score of changes across GH and HF and the number of cross-platform contributors, with the highest proportion of multiple-author contributions (44.9%) observed in PTLM families with high similarity scores, compared to 29.0% in those with low similarity scores. The analysis of the distribution of single and multiple cross-platform contributors across different similarity score categories between GH and HF (Figure 8) shows a clear trend. For low similarity scores, 37.5% of PTLM families have a single cross-platform author, while 29.0% have multiple authors. In PTLM families with moderate similarity scores, the proportion of single-author contributions decreases to 33.6%, and multiple-author contributions also decrease to 26.1%. In PTLM families with high similarity scores, however, single-author contributions drop to 28.9%, while multiple-author contributions increase significantly to 44.9%, the highest proportion observed.

A Chi-Square test of independence confirmed a statistically significant relationship between contributor composition and similarity score category (χ^2 = 6.38, p = 0.0412). The effect size, measured using Cramér's V, was 0.140, indicating a moderate association. While the test does not imply causality, the observed distribution suggests that PTLM families with high alignment in change types across platforms are more frequently associated with multiple cross-platform contributors. This association may reflect how shared authorship contributes to or co-occurs with more coordinated development practices across GH and HF.

Summary

Approximately 77% of PTLM families show moderate to high similarity in commit change types across GH and HF, with the degree of similarity associated with model maturity and the presence of cross-platform contributors.

4.2 **RQ$_2$:** What are the synchronization patterns of commit activities across PTLMs on upstream and downstream?

4.2.1 Motivation

In RQ1, our analysis revealed that GH and HF exhibit distinct types of changes in their commit activities, with clear differences in focus and scope. While some thematic overlap exists, the nature, distribution, and intensity of these commit change types differ significantly between the two platforms. However, the extent to which commit activities are synchronized across the platforms remains unclear. Understanding how commit activities in PTLM families synchronize over time across both platforms is important for streamlining the release process and optimizing integration between GH and HF. To address this, we investigate the key characteristics of overlapping commit activities that form synchronization patterns (see example in Figure 9) and the extent to which synchronization occurs across the two platforms. By identifying these patterns, our goal is to better understand how activities are synchronized across GH and HF repositories.

4.2.2 Approach

4.2.2.1 Identifying and explaining the factors contributing to understanding the synchronization patterns. To understand the factors contributing to the synchronization patterns of commit activities across HF and GH, we analyzed the combined commit histories of 325 PTLM families. We manually examined 177 families using a combination of open and closed card sorting methods (Wood and Wood, 2008), and automatically labeled the remaining 148 using a custom script. In the open card sorting phase, we analyzed 50 randomly sampled families to inductively develop categories related to delay patterns, commit frequency, and synchronization types, based on visualizations of commit timelines. These categories were then applied in a closed card sorting phase to the remaining 127 manually reviewed families. The remaining 148 families were categorized using an automated method based on the established scheme. The following steps outline our approach:

Step 2.1: Selection of representative samples. We used a stratified random sampling method[42] with a 95% confidence level and a 5% margin of error (Singh and Masuku, 2014, Cocks and Torgerson, 2013) to select 177 PTLM families as representative samples from a population of 325 PTLM families for manual analysis.

Step 2.2: Visualizing the representative samples. To understand the synchronization of commit activities between GH and HF for each of the 177 PTLM families, we generated visualizations using scatter plots, plotting the dates of commits on both platforms. We aggregated these activities into bi-weekly intervals (2 weeks are coalesced together) to assess the synchronization of commit activities and identify patterns in their synchronization or divergence. The choice of a bi-weekly window strikes a balance between capturing significant activity trends without overloading the visualization with too much detail. A longer window, such as a monthly

[42] https://www.surveymonkey.com/mp/sample-size-calculator/

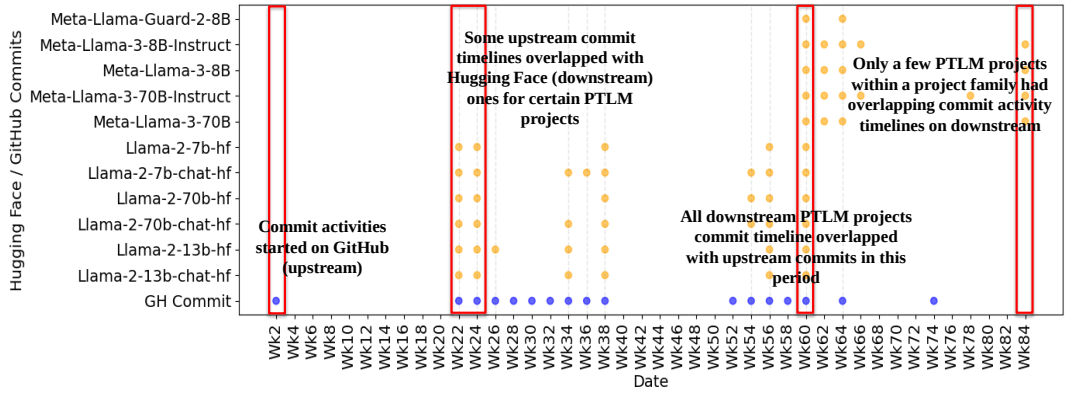

Fig. 9 Visualization of commit activity patterns for a PTLM family, illustrating activity overlap, synchronization types, and intensity across the upstream (GH) and downstream (HF) platforms. The blue dots represent biweekly commit activities on GitHub, while the yellow dots indicate the biweekly commit activities of different PTLM variants within the same family.

one, would risk smoothing out important synchronization events, while a shorter window, like daily or weekly, could result in overly cluttered plots that are hard to interpret, especially for PTLM families with long lifespans.

Vertical dashed lines (see example in Figure 9) were added to the plots at the start of each bi-weekly period when at least one commit activity occurred on both GH and HF. This helped us visualize the co-occurrence of activities across both platforms and identify any synchronization or divergence in the timing of commit activities.

Step 2.3: Selection and coding of 50 random samples. Initially, the first and second authors independently analyzed the visualization of 50 randomly selected PTLM families from the 177 projects to identify factors contributing to the observed synchronization patterns. The process to identify these factors involved:

1. Assigning descriptive phrases to observations derived from scatter plots that visualized commit activity relationships between GH and HF, helping surface underlying synchronization patterns (see Table 1 for examples. Columns two and three show how each author described their observations.);
2. Comparing, refining, and consolidating these descriptive phrases into a shared set of preliminary codes that captured synchronization characteristics (see columns four to eight in Table 1 for examples of intensity and synchronization types, the given codes, interpretations, and final pattern names);
3. Calculating inter-rater agreement between the authors for columns 4 to 7 using Cohen's Kappa, which yielded a score of 0.73—indicating substantial agreement. This outcome provided a foundation for the closed card sorting phase to consistently categorize synchronization patterns across the remaining dataset.

Step 2.4: Coding of the Remaining Dataset. After confirming inter-rater reliability in the previous step, the first author used a closed card sorting approach to code the remaining 127 sampled PTLM families based on the established categories. The substantial agreement observed in the initial phase, reflected by a Cohen's Kappa score of 0.73, justified this method. Studies with Kappa scores 0.61 and above commonly permit single-rater coding in similar contexts (El Emam, 1998). Furthermore, no new codes emerged during this phase, demonstrating consistency with the predefined framework and ensuring a uniform identification of synchronization patterns across the dataset.

Step 2.5: Naming of identified patterns. Following the coding of the remaining dataset, the authors convened to assign appropriate names to the identified codes. This process was guided by three key aspects: (1) activity lead, which determines which platform exhibited activity first within a bi-weekly period (for example, Figure 9 shows that activities commenced on GH in week 2, while none of the HF PTLMs had commit activities until week 22); (2) synchronization types, which assess the overlap in activity periods between the two platforms (for example, in Figure 9, weeks 22 and 24 show that more than 50% of the PTLMs in the family had commit

Table 1: Examples of descriptive observations and consolidated codes from independent author analysis of synchronization patterns across HF and GH. Sync: Synchronization, Act: Activity, CS: Complete Synchronization, PS: Partial Synchronization, AS: Asynchronous.

Family	Author 1	Author 2	Act. Lead	Intensity	Sync Types	Code	Meaning of Code	Pattern Name
hkunlp_instructor	HF has $\leq$3 biweekly activities but all align with GH. GH has longer activities period but sporadic in nature	Frequent sporadic GH with rare HF	None	R	CS	RCS	Rare complete synchronization	Rare synchronization
benjamin_segment	HF and GH activities are irregular and partially aligned	Frequent sporadic GH with sporadic HF	HF	S	PS	SPS	Sporadic partial synchronization	Disperse synchronization
meta-llama_CodeLlama	HF has $\leq$3 biweekly activities that are not aligned with GH. GH extended for a long period but with bursts of activities at times	Sporadic GH with rare HF	HF	R	AS	RAS	Rare Asynchronous	Rare Disjoint Pattern

activity timelines overlapping with GH, followed by a burst of activity from more than 90% of the PTLMs on HF lasting for 8 weeks. Meanwhile, some commit activities from HF PTLMs do not overlap with the GH timeline, such as in weeks 66 and 78. This suggests that the family exhibits partial synchronization); and (3) intensity of commit activity updates, which captures the concentration and propagation of updates across PTLM models within a PTLM family (for example, in Figure 9, after four weeks of overlapping commit activity between GH and some HF PTLMs, there is a burst of activity lasting 8 weeks from more than 90% of the PTLMs, followed by sporadic activity and then a 12-week period (week 40 to 54) of inactivity on both platforms). This clearly indicates sporadic intensity, as the overlapping periods are not continuous). We combine two of these three aspects—synchronization type and intensity—to define a specific synchronization pattern for each family, as exemplified in Table 1. The third aspect, activity lead, was not used to name the patterns but served as an independent indicator, helping us understand which platform typically initiates development activity within each synchronization pattern.

Step 2.6: Automatic coding and naming of the remaining samples. To extend our analysis beyond the manually coded 177 samples, we developed a script (available in our replication package (Adekunle, 2025)) that processes commit histories from both platforms and maps them to their respective PTLM families. It merges datasets, filters out manually codeed PTLM families while retaining relevant ones, standardizes timestamps to a common format, and organizes commit activities into biweekly periods for structured analysis.

- **Identifying the activity lead:** The script identifies where commit activities first occurred in the initial biweekly period containing commit activity. For example, Figure 11 shows a case where activity on the upstream (GH) repository leads for more than 52 weeks before any activity appears on the downstream (HF) repository. By segmenting commit histories and analyzing update sequences across platforms, we determine whether commit activity starts on GH, HF, or both simultaneously. We detail the algorithm for identifying activity lead in Appendix B.1.
- **Identifying overlapping commit activity timelines:** Our script categorizes recurring commit trends based on how updates progress over time. We examine whether (1) commit activities consistently appear on both platforms within the same periods, (2) their occurrences align in some instances but not others, (3) commit activities are concentrated on one platform with minimal influence from the other, or (4) commit activities on the two platforms lack a clear relationship. An example of frequent activity on both platforms can be

found in Figure 12. We have detailed the algorithm for identifying the overlapping period of activities in Appendix B.2.

– **Identifying the frequency of commit activities timeline:** We evaluate how frequently commit activities overlap across both platforms. We determine whether (1) commit activities on HF occur only in a few scattered periods, (2) updates on both platforms align consistently over an extended period, or (3) commit activity appears irregular without a clear trend. Figure 12 also illustrates an example where commit activities on both platforms overlap consistently over a long period. We also gave a full details of algorithm used to identify the intensity activities between GH and HF in Appendix B.3.

Step 2.7: Validating and applying the automated coding script: We validated the coding script by using it to re-code the 177 PTLM families that we had manually annotated (see columns 5–7 of Table 1 for examples of codes). The script achieved Cohen's Kappa scores of 1.00 for lag (delay) (example in column 4 of Table 1) and 0.96 for pattern codes (example in column 7 of Table 1). During coding, the script misclassified three PTLM families by labeling a "*sporadic intensity* (S)" PTLM family as "*rare intensity* (R)" (example of intensity in column 5 of Table 1). These misclassifications occurred due to fluctuations in commit activity that fell near the boundaries between defined intensity categories. These boundaries were operationalized using rules based on the duration and pattern of commit overlaps between GH and HF. Specifically, a family was labeled Rare (R) if all variants exhibited commit activity within at most three biweekly periods. A label of Frequent (F) was assigned if any variant showed at least five consecutive biweekly periods of overlap with GitHub activity. Cases that did not meet either condition were categorized as Sporadic (S), reflecting moderate or inconsistent overlap patterns. These thresholds were established based on empirical observations made during the manual annotation phase. Subsequently, we manually reviewed and corrected the three misclassified cases. After validation, we applied the script to code the commit events of the remaining 148 PTLM families.

After establishing these characteristics, we assign a meaningful name (see example in column 9 of Table 1) to each project based on how commit activities unfold across platforms. These names correspond to distinct commit trends and provide insights into how updates are structured over time.

4.2.2.2 Explaining the identified synchronization patterns. After completing the coding process, we treated the codes derived from activity lead, synchronization types, and intensity of PTLM updates as dimensions defining the synchronization patterns of commit activities between GH and HF. These dimensions were chosen because together they capture key temporal dynamics and distributional characteristics of commit activity across platforms, such as where activities typically begin, the extent of temporal overlap between platforms, and the frequency of switching between them. These three dimensions capture the core synchronization behaviors of upstream and downstream platforms. We later illustrate the resulting patterns with examples of PTLM families that exhibit these synchronization behaviors. This analysis not only classifies the patterns but also deepens our understanding of release workflows. Detailed explanations and examples appear in Section 4.2.3.

4.2.3 Results.

There are three codes for intensity, four codes for synchronization types, and three codes for lag (delay). Since intensity and synchronization types were combined to form synchronization patterns, some combinations are theoretically implausible or absent in practice. For example, the asynchronous (AS) code lacks overlapping activity, making combinations such as (F, AS, CS) unlikely, while the cross-variant synchronization (VS) typically appears as sporadic and simultaneous. We report only patterns that are empirically grounded in observed commit synchronization. Table 2 summarizes these characteristics.

intensity: This characteristic captures how often commit activities occur across GH and HF. We identified three levels—*rare*, *sporadic*, and *frequent*—which are illustrated below with examples.

– **Rare Intensity:**
This occurs when commit activities are infrequent on one or both platforms, often due to model stability, limited updates, or lack of maintenance interest. Examples: MBZUAI/LaMini-Flan-T5-783M (HF[43] vs

[43] https://huggingface.co/MBZUAI/LaMini-Flan-T5-783M

Table 2: synchronization pattern characteristics in PTLMs: Intensity (frequency of activity), synchronization type (overlapping of activities), and Lag (delay) (order of updates)

Intensity	**Synchronization type**	**Lag (delay)**
Complete Synchronization (CS)	Rare (R)	GH First
Partial Synchronization (PS)	Frequent (F)	HF First
Cross-variant Synchronization (VS)	Sporadic (S)	Simultaneous Activities
	No Synchronization (AS)	

GH[44]) and cais/HarmBench-Llama-2-13b-cls (HF[45] vs GH[46]).

– **Sporadic Intensity:**
Characterized by irregular bursts of updates on one platform while the other remains stable. This reflects sprint-based development and platform-specific priorities. Examples: alisawuffles/roberta-large-wanli (HF[47] vs GH[48]), Babelscape/wikineural-multilingual-ner (HF[49] vs GH[50]), aubmindlab/bert-base-arabertv02 (HF[51] vs GH[52]).

– **Frequent Intensity:**
Involves consistent updates across both platforms, often sustained over multiple periods. This indicates tight synchronization, shared contributors, and mechanisms such as synchronized deployment cycles, automated update pipelines, or coordinated development workflows. Examples: 01-ai/Yi-1.5-34B-Chat (HF[53] vs GH[54]), BAAI/llm-embedder (HF[55] vs GH[56]).

Synchronization Type: This describes how commit activities on GH and HF align over time, reflecting the synchronization of activities across platforms. Inspired by agile practices like dependency management and team synchronization (e.g., daily meetings, sprint planning) (Strode et al., 2012), we identified four synchronization types: complete, partial, cross-variant, and none.

– **Complete synchronization (CS):**
Commit activities are consistently time-aligned across GH and HF, showing deliberate synchronization—e.g., code updates on GH and documentation on HF within the same period. Examples: MBZUAI/LaMini-Flan-T5-783M (HF[57], GH[58]), 01-ai/Yi-1.5-34B-Chat-16K (HF[59], GH[60]), and cais/HarmBench-Llama-2-13b-cls (HF[61], GH[62]).

[44]https://github.com/mbzuai-nlp/lamini-lm
[45]https://huggingface.co/cais/HarmBench-Llama-2-13b-cls
[46]https://github.com/centerforaisafety/HarmBench
[47]https://huggingface.co/alisawuffles/roberta-large-wanli
[48]https://github.com/alisawuffles/wanli
[49]https://huggingface.co/Babelscape/wikineural-multilingual-ner
[50]https://github.com/Babelscape/wikineural
[51]https://huggingface.co/aubmindlab/bert-base-arabertv02
[52]https://github.com/aub-mind/arabert
[53]https://huggingface.co/01-ai/Yi-1.5-34B-Chat
[54]https://github.com/01-ai/Yi
[55]https://huggingface.co/BAAI/llm-embedder
[56]https://github.com/FlagOpen/FlagEmbedding
[57]https://huggingface.co/MBZUAI/LaMini-Flan-T5-783M
[58]https://github.com/mbzuai-nlp/lamini-lm
[59]https://huggingface.co/01-ai/Yi-1.5-34B-Chat-16K
[60]https://github.com/01-ai/yi
[61]https://huggingface.co/cais/HarmBench-Llama-2-13b-cls
[62]https://github.com/centerforaisafety/HarmBench

- **Partial synchronization (PS):**
  Commit activities align at some points but diverge at others, reflecting intermittent synchronization due to shifting priorities or platform focus. Examples: bigcode/starcoder2-15b (HF[63], GH[64]), google-bert/bert-large-uncased-whole-word-masking (HF[65], GH[66]), and dbmdz/bert-base-german-cased (HF[67], GH[68])).
- **Cross-variant synchronization (VS):**
  Synchronization occurs across multiple HF model variants but are not temporally aligned with the corresponding GH repository, reflecting internal HF synchronization independent of GH updates. Examples: google/switch-base-128 (HF[69], GH[70]), intfloat/multilingual-e5-large (HF[71], GH[72]), and jinaai/jina-embeddings-v2-base-en (HF[73], GH[74]).
- **No synchronization (AS):**
  Activities on GH and HF, or across HF variants, operate independently without alignment, possibly due to different maintainers or missing linked repositories. Examples: CAMeL-Lab/bert-base-arabic-camelbert-mix-ner (HF[75], GH[76]), csebuetnlp/banglat5_nmt_bn_en (HF[77], GH[78]), and dandelin/vilt-b32-mlm (HF[79], GH[80]).

Lag (delay): This refers to the order in which PTLM commit activities first appear on GH and HF. A lag occurs when updates appear on one platform significantly earlier than the other, creating a temporal gap. We identified three types of workflow lag:

- **GH First:** Here, model-related commits begin on GH before being uploaded to HF. This mirrors software workflows where code is first developed upstream before distribution. In PTLM development, GitHub typically hosts training routines or bug fixes, while HF is used later for broader accessibility. For instance, google-bert/bert-large-uncased-whole-word-masking was released on GH[81] in 2019 and appeared on HF[82] only in 2022. Similar patterns were found for dbmdz/bert-base-german-cased (HF[83] vs GH[84]) and tner/roberta-large-ontonotes5 (HF[85] vs GH[86]).
- **HF First:** In this pattern, models are published on HF before any related GitHub activity. While the initial development may occur privately or internally, HF provides early public access. Code appears later, often due to internal release policies. For example, Efficient-Large-Model/VILA1.5-13b appeared first on HF[87],

[63] https://huggingface.co/bigcode/starcoder2-15b
[64] https://github.com/bigcode-project/starcoder2
[65] https://huggingface.co/google-bert/bert-large-uncased-whole-word-masking
[66] https://github.com/google-research/bert
[67] https://huggingface.co/dbmdz/bert-base-german-cased
[68] https://github.com/dbmdz/berts
[69] https://huggingface.co/google/switch-base-128
[70] https://github.com/google-research/t5x
[71] https://huggingface.co/intfloat/multilingual-e5-large
[72] https://github.com/intfloat/SimKGC
[73] https://huggingface.co/jinaai/jina-embeddings-v2-base-en
[74] https://github.com/jina-ai/finetuner
[75] https://huggingface.co/CAMeL-Lab/bert-base-arabic-camelbert-mix-ner
[76] https://github.com/CAMeL-Lab/CAMeLBERT
[77] https://huggingface.co/csebuetnlp/banglat5_nmt_bn_en
[78] https://github.com/csebuetnlp/banglanmt
[79] https://huggingface.co/dandelin/vilt-b32-mlm
[80] https://github.com/dandelin/ViLT
[81] https://github.com/google-research/bert
[82] https://huggingface.co/google-bert/bert-large-uncased-whole-word-masking
[83] https://huggingface.co/dbmdz/bert-base-german-cased
[84] https://github.com/dbmdz/berts
[85] https://huggingface.co/tner/roberta-large-ontonotes5
[86] https://github.com/asahi417/tner
[87] https://huggingface.co/Efficient-Large-Model/VILA1.5-13b

with a bulk GitHub commit added later[88]. Similarly, LumiOpen/Poro-34B was on HF[89] a full year before appearing on GH[90]. In contrast, MBZUAI/LaMini-Flan-T5-783M had a short delay between the platforms (HF[91] vs GH[92]).

– **Simultaneous Activity:** Commits occur on both platforms within about two weeks, showing no strong order. Updates may differ but reflect synchronized activity. This pattern may result from agile workflows or automation pipelines that coordinate releases across platforms. Examples include 01-ai/Yi-1.5-34B-Chat-16K (HF[93] vs GH[94]), bigcode/starcoder2-15b (HF[95] vs GH[96]), and cais/HarmBench-Llama-2-13b-cls (HF[97] vs GH[98]).

Summary

We identified three aspects of cross-platform commit behavior—lag, synchronization type, and intensity. Lag measures the time delay between corresponding commits, synchronization type evaluates how activities on GH and HF align over time, and intensity assesses the frequency and consistency of commits on both platforms. Together, these aspects capture key elements of workflow synchronization, reflecting practices that contribute to efficient PTLM release strategies

We identified eight distinct synchronization patterns in commit activities during the release process of PTLMs, based on the two characteristics defined earlier. These patterns reflect varying levels of synchronization between activities across both ecosystems. Each pattern is illustrated with examples from development activities and is presented in order—beginning with complete synchronization, followed by partial synchronization, and concluding with those that show no synchronization at all.

Rare Synchronization Pattern (i.e., Rare complete synchronization (RCS))

Definition: This pattern represents *rare but complete synchronization*, where one platform exhibits infrequent commit activity—typically across only two to three periods—while the other maintains a more continuous or extended activity timeline. Despite the disparity in the frequency of commits, each commit on the less active platform is completely overlapped periodically with commits on the more active one.
Explanation: This pattern may result from several factors beyond deliberate synchronization. Developers might prioritize one platform—often GH for core development—while treating HF as a secondary publishing endpoint, pushing only essential updates. Alternatively, the pattern could stem from limited contributor availability or a lack of ownership across platforms, possibly reflecting a passive or indifferent maintenance approach. While synchronization is technically complete when it happens, its rarity raises significant release engineering concerns: users may unknowingly rely on outdated or incomplete models, reducing reproducibility and potentially undermining trust in the model's reliability and support.
Notable Instances: Examples of this pattern can be found in these PTLM family in our analysis, Tner Roberta, Lnyuan Distilbert, and Medicalai ClinicalBert PTLM family, with one example visualized in Figure 10. These PTLM families demonstrate that, despite the rarity of synchronization, each commit activity in one platform is fully aligned with activities in the other, reflecting coordinated but *sparse synchronization*.

[88] https://github.com/NVLabs/VILA
[89] https://huggingface.co/LumiOpen/Poro-34B
[90] https://github.com/LumiOpen/evaluation
[91] https://huggingface.co/MBZUAI/LaMini-Flan-T5-783M
[92] https://github.com/mbzuai-nlp/lamini-lm
[93] https://huggingface.co/01-ai/Yi-1.5-34B-Chat-16K
[94] https://github.com/01-ai/yi
[95] https://huggingface.co/bigcode/starcoder2-15b
[96] https://github.com/bigcode-project/starcoder2
[97] https://huggingface.co/cais/HarmBench-Llama-2-13b-cls
[98] https://github.com/centerforaisafety/HarmBench

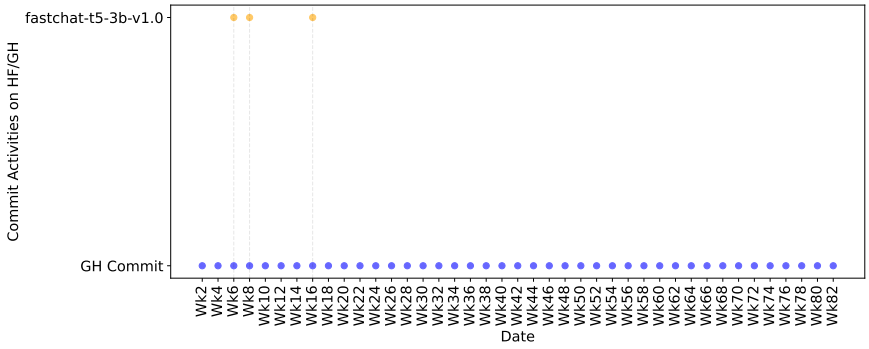

Fig. 10 Visualization of one of the families in the Rare Synchronization Pattern. The yellow dots show rare commit activities of a PTLM family on HF, while the blue dots represent frequent commit activities on GH.

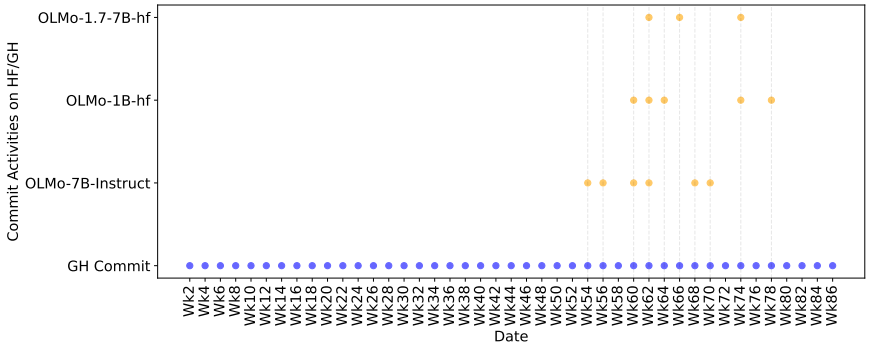

Fig. 11 Visualization of one of the families in the Intermittent Synchronization Pattern. The yellow dots represent sporadic and delayed (lag) commit activities on HF, while the blue dots indicate frequent commit activities on GH.

Intermittent Synchronization Pattern (i.e., sporadic complete synchronization (SCC))

Definition: The Intermittent Synchronization Pattern is characterized by commit activities occurring at irregular intervals across five or more periods on either HF or GH, where each activity is *completely coordinated* with corresponding commit activities on the other platform. This pattern differs from the *Rare Synchronization Pattern* due to a longer duration of activity and greater intensity of synchronization across platforms.
Explanation: This pattern reflects irregular but recurring synchronization. Commit activities on HF span five or more periods but are spaced unevenly, often aligning fully with GH updates when they occur. This may suggest that contributors update HF in bursts—perhaps following internal milestones or after bundling multiple changes. Alternatively, it could reflect loosely structured workflows, where contributors act independently with minimal synchronization planning. Though synchronization exists, the irregular timing may still delay important updates, leading to possible version drift. For release engineering, this poses moderate risk—users may not receive updates promptly, impacting consistency and deployment accuracy.
Notable Instances: Examples of PTLM PTLM family exhibiting the Intermittent Synchronization Pattern in our study include Facebook ESM2, EleutherAI GPT, and Unsloth Models. A typical example of this pattern is visualized in Figure 11, showing how synchronization is maintained, though irregular, between the commit activities on HF and GH.

Frequent Synchronization Pattern (i.e., frequent complete synchronization (FCC))

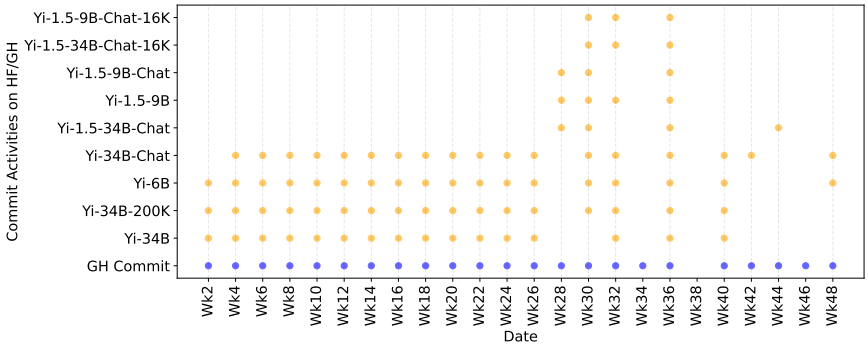

Fig. 12 Visualization of one of the families in the Frequent Synchronization Pattern. The yellow dots represent frequent commit activities from at least three PTLM families on HF and GH, indicating complete synchronization between the two platforms.

Definition: This pattern is characterized by regularly occurring commit activities on both HF and GH for at least three PTLMs within a PTLM family, extending over five or more periods, with complete synchronization maintained throughout.
Explanation: This pattern reflects a highly disciplined and well-structured release practice. Commit activities on HF occur consistently across five or more consecutive periods, maintaining full synchronization with those on GH. Such regular synchronization could indicate a mature project workflow—possibly driven by strong collaboration among contributors, automated CI/CD pipelines, or well-defined release schedules. In projects with multiple PTLMs, the alignment across all PTLMss in the same period further emphasizes deliberate, synchronized effort. For release engineering, this pattern is ideal: it ensures users can reliably access up-to-date versions across both platforms, reducing confusion, version drift, and maintenance ambiguity. It also boosts confidence in the model's reliability and long-term support.
Notable Instances: Examples of PTLM PTLM family exhibiting the Frequent Synchronization Pattern in our study include BAAI_BGE_models and Maidalun1020 Models. A typical example is visualized in Figure 12, where synchronization between the PTLM PTLM family commit activities on HF and GH is maintained regularly over multiple periods.

Disperse Synchronization Pattern (i.e., sporadic partial synchronization (SPC))

Definition: The Disperse Synchronization Pattern is characterized by the partial synchronization of commit activities between GH and HF, where some commit activities on one platform occur simultaneously with those on the other, while other commit activities take place independently. This synchronization pattern often involves commit activities on HF extending beyond the active period of commit activities on GH.
Explanation: This synchronization pattern reflects a transitional or handoff-like behavior between platforms. In many cases, one platform—typically GH—initiates commit activities and maintains them for several periods. As activity on GH declines or ceases, commit activities begin or intensify on HF. The overlap between the two platforms is often brief and limited to only a few PTLMs, with the majority of HF activity occurring independently after GH activity has ended. This raises practical concerns: if changes made on GH during its active period are not fully replicated on HF, users accessing the models later may receive incomplete or outdated versions. Additionally, the source and nature of the updates on HF—whether performed locally, manually, or from external automation—remain unclear, especially when there's no ongoing synchronization with GH. This partial and time-shifted synchronization may stem from strategic migration, contributor preference, or toolchain differences, but it risks fragmenting the model's update history and undermining reproducibility.
Notable Instances: Examples of families exhibiting the Disperse Synchronization Pattern in our study include Cardiffnlp Ttweeteval, Facebook Galactica, and Meta Llama. A typical example is visualized in Figure 13, where the synchronization and independence of commit activities across platforms are evident.

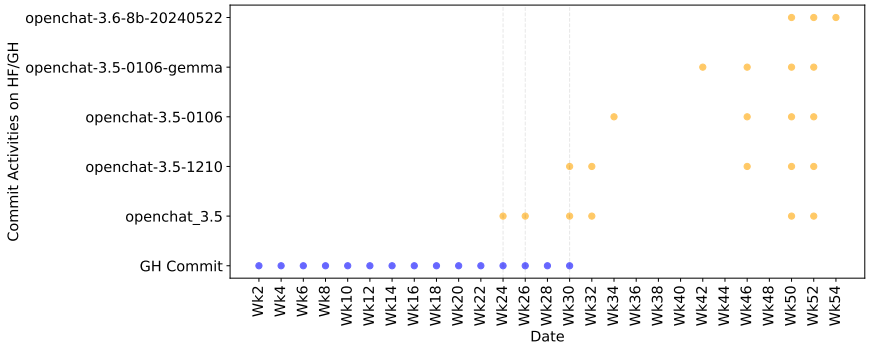

Fig. 13 Visualization of one of the families in the Disperse Coordination pattern. The yellow dots represent commit activities on HF, while the blue dots represent commit activities on GH. The visualization shows that only a few PTLMs in the family periodically overlap with GH, while others commence their commit activities after GH has ceased activity.

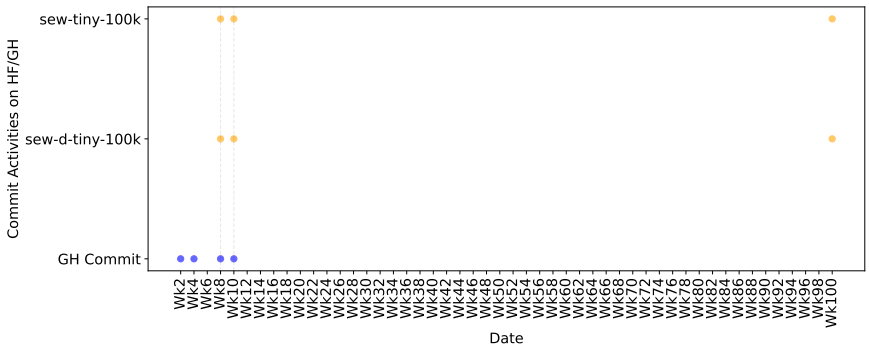

Fig. 14 Visualization of one of the families in the Sparse Synchronization Pattern. The yellow dots represent rare commit activities (lasting ≤ 3 periods) on HF, while the blue dots represent sporadic commit activities (spanning > 3 periods) on GH.

Sparse Synchronization Pattern (i.e., rare partial synchronization (RPC))

Definition: This pattern is characterized by brief periods of commit activities on either platform, typically lasting no more than three periods per PTLM. These commit activity periods are spread over a long duration, with extended gaps between them.

Explanation: This pattern reflects rare and short-lived commit activities across platforms, typically lasting no more than three periods per PTLM, and often spread out with long gaps in between. Commit activities on GH may occur early and infrequently, while HF may exhibit a few rare updates, sometimes appearing only in the final periods. This temporal separation could suggest that the model development was done locally, with contributors pushing finalized code to GH and later uploading the model to HF. In such cases, updates on HF may involve post-release tasks such as documentation or minor adjustments that do not require corresponding changes on GH. While this pattern may seem suboptimal from a release engineering standpoint—due to the limited synchronization and visibility—it may not necessarily indicate poor practices if the development process is well-managed behind the scenes. However, the lack of consistent synchronization and the brevity of activity periods could hinder reproducibility and traceability for end users.

Notable Instances: Examples of families exhibiting this pattern in our study include Prithivida Parrot, EmergentMethods Gliner, and Wukevin TCR-BERT. A typical example is visualized in Figure 14.

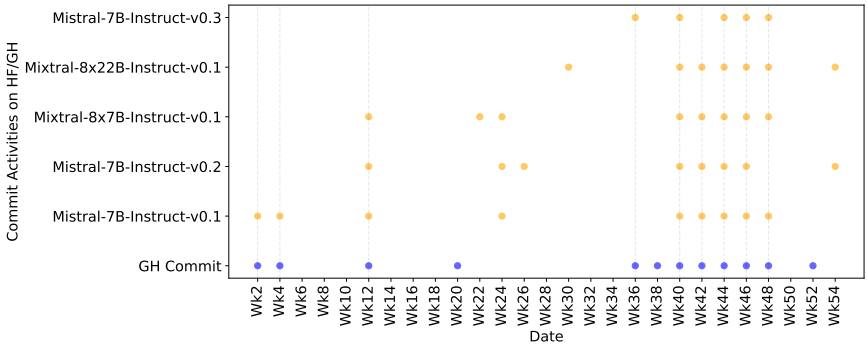

Fig. 15 Visualization of one of the PTLM families in the Dense Partial Synchronization Pattern

Dense Partial Synchronization Pattern (i.e., frequent partial synchronization (FPC))

Definition: This pattern is characterized by two distinct phases of commit activities: a phase of scattered and sporadic commit activities with limited synchronization with commit activities on the other platform, followed by a phase of at least five consecutive periods of frequent and coordinated commit activities. These phases can also occur in reverse order.

Explanation: This pattern reflects a dynamic shift in the synchronization of commit activities between platforms. In the early phase, commit activities are scattered and sporadic, with limited or no alignment across platforms. This may reflect a lack of centralized planning, too many loosely involved contributors, or experimentation without a structured release process. Later, the project enters a more stable phase with frequent and well-coordinated commits on both platforms, sustained over at least five periods. This shift may result from a reduced contributor base, clearer team roles, or the adoption of a more structured release strategy. In some cases, the order is reversed—starting with strong synchronization and later becoming less structured, possibly due to reduced interest or changing priorities. Unlike the Sparse Synchronization Pattern, which shows brief and isolated activity, or the Disperse Synchronization Pattern, where one platform continues after the other, this pattern is marked by a clear transition between sporadic and coordinated phases. It highlights how synchronization practices can evolve as projects mature or as team dynamics shift.

Notable Instances: Examples of families following this pattern in our study include BigScience Workshop, Mistral-Inference, and JackFram Llama. A typical example of this pattern is visualized in Figure 15.

Sporadic Disjoint Pattern (i.e., Sporadic asynchronous (SAS))

Definition: The Sporadic Disjoint Pattern is characterized by commit activities occurring at different periods on each platform, with no synchronization between commit activities on GH and commit activities on HF.

Explanation: This pattern reflects a lack of synchronization in commit activities between GH and HF. Commits appear first on one platform and later on the other, often in alternating phases without overlapping periods. This may suggest a sequential workflow where development or updates are first completed and stabilized on one platform—often HF—and only later pushed to GH, possibly after internal testing or review. Such behavior could stem from workflow preferences, platform-specific roles, or a manual syncing process. While this approach allows for flexibility and separation of concerns, it carries risks such as version drift or delayed updates across platforms. From a release engineering perspective, it isn't inherently flawed, but it may indicate inefficiencies or a lack of automation in maintaining consistency. The absence of overlap may also point to limited resources, fragmented teams, or unintegrated tools. Unlike other patterns, this one shows strictly alternating and non-overlapping activity phases, suggesting a reactive or staged update strategy rather than continuous integration across platforms.

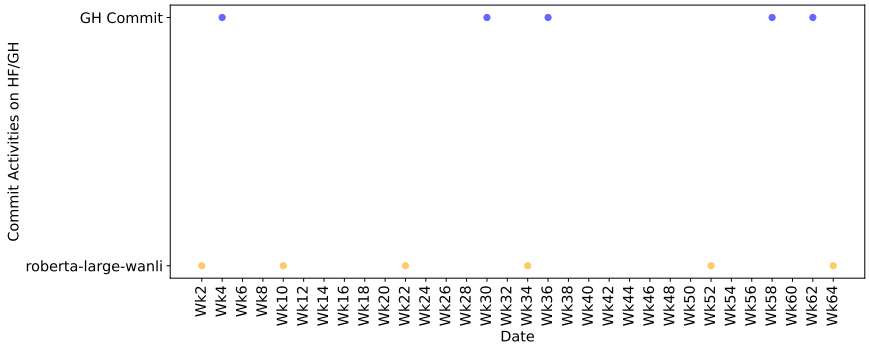

Fig. 16 Visualization of one of the PTLM families in the Sporadic Disjoint Pattern

Notable Instances: Examples of PTLM families exhibiting this pattern include Microsoft BiomedNLP, Alisawuffles Roberta, and Facebook Hubert. A typical example of this pattern is visualized in Figure 16.

Rare Disjoint Pattern (i.e., rare asynchronous (RAS))

Definition: This pattern is characterized by commit activities on one platform being very rare, occurring in no more than three periods per PTLM, and showing no synchronization with commit activities on the other platform.

Explanation: This pattern shows that commit activities on one platform are very limited and completely uncoordinated with those on the other. In some cases, one platform finishes all its activity before the other even begins. If commit activity appears first on HF, it may suggest that models were trained and uploaded before the training code or supporting resources were made available on GH. Conversely, if GH activity comes first, it could mean the training code was shared and stabilized before any model was uploaded to HF. This sequential and uncoordinated process could be risky from a release engineering standpoint. Users might access the model before understanding how it was built, or access the training code without access to the corresponding model, leading to inconsistencies, confusion, or trust issues. While not necessarily wrong, this approach lacks transparency and may reflect poor release planning or a fragmented development process. In contrast to patterns with even minimal synchronization, this one highlights a complete disconnect between platforms, suggesting missed opportunities for integrated, reliable, and user-aligned releases.

Notable Instances: Examples of PTLM families exhibiting this pattern in our study include Facebook Roberta, Google Muril, and Meta-llama CodeLlama. A typical example of this pattern is visualized in Figure 17.

Summary

We identified eight distinct synchronization patterns in commit activities of PTLMs on HF and GH: *Dense partial synchronization*, *Disperse synchronization*, *Frequent synchronization*, *Intermittent synchronization*, *Rare synchronization*, Rare disjoint, *Sparse synchronization*, and *Sporadic disjoint*, which help practitioners understand how commit activities align or diverge across platforms, which help practitioners understand how commit activities align or diverge across platforms, enabling them to identify synchronization challenges and optimize workflows for smoother PTLM development across HF and GH.

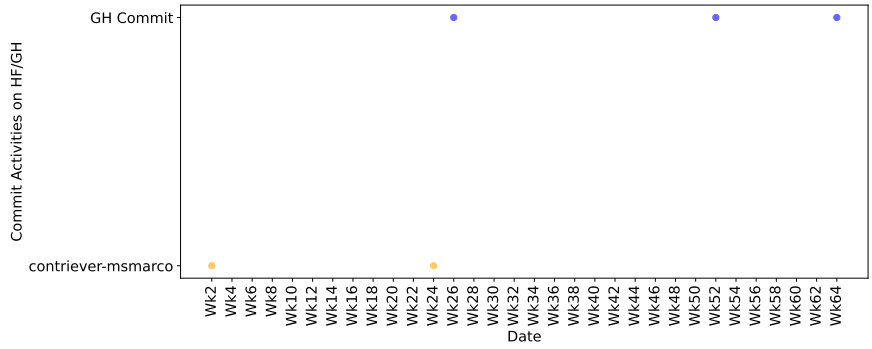

Fig. 17 Visualization of one of the families in the Dense synchronization pattern

4.3 **RQ$_3$:** How are the synchronization patterns between upstream and downstream distributed across PTLM families?

4.3.1 Motivation

In RQ2, we identified distinct synchronization patterns that describe how PTLM families manage commit activities across GH and HF. To understand how these synchronization patterns are distributed and evolve across PTLM families of different ages, we first examine the distribution of these patterns within PTLM families of varying maturity. This analysis helps practitioners assess whether their current coordination approach is aligned with common patterns across similar PTLMs, and consider adjustments to improve cross-platform consistency, responsiveness, or contributor alignment over time. We then explore how these patterns evolve over time in the early and later stages of PTLM families, particularly focusing on how younger and older PTLM families transition between synchronization patterns. Additionally, we investigate whether the number of contributors and commit activities are associated with these transitions. Understanding these dynamics can help model owners anticipate synchronization trends and reflect on the weaknesses in their current practices, potentially forming a recommendation on where to improve synchronization or documentation efforts. It may also help platform maintainers support more effective synchronization tools and enable researchers to investigate less examined aspects of multi-platform release behavior.

4.3.2 Approach

4.3.2.1 Exploring the relationship between change types and synchronization pattern on GH and HF. To analyze the relationship between synchronization patterns and change types, we processed data separately for GitHub and Hugging Face. For GitHub, we merged the labeled commit data with synchronization pattern information based on PTLM family names as explained in RQ1. Since each PTLM family is associated with a single synchronization pattern, we grouped the merged data by synchronization pattern and change type, and counted the number of unique commit messages. This resulted in a matrix, where values represent the proportion of each change type within a given pattern. These proportions were visualized using a heatmap to illustrate how change types are distributed across synchronization patterns on GitHub.

For Hugging Face, we adopted a complementary model-level approach to account for the fact that each PTLM family can contain multiple models. We calculated the distribution of change types per model by normalizing the change type counts within each model. This was followed by a merge with synchronization pattern information and an aggregation step where we computed the median proportion of each change type across all models within a pattern. This median-based aggregation mitigates the influence of outlier models that may dominate the activity of a PTLM family. We visualized the resulting matrix using a log-scaled heatmap to highlight the range of proportions while displaying the original values as annotations.

To statistically assess whether the distribution of change types significantly differs between GitHub and Hugging Face within each synchronization pattern, we grouped the original 15 fine-grained change types into four broader categories based on the classification proposed by Bhatia et al. (2023): maintenance (e.g., preprocessing, parameter tuning, model structure), meta program (e.g., documentation, sharing, validation infrastructure), data category (e.g., input/output data, project metadata), and dependency management (e.g., adding, removing, updating dependencies). This grouping was necessary to reduce dimensionality and ensure the robustness of our statistical tests. Using a 2×15 contingency table per pattern would likely lead to sparse cells, violating the assumptions of the chi-square test. Aggregating into four higher-level categories helped maintain sufficient expected frequencies per cell, thereby improving the reliability, statistical validity, and interpretability of the results. For each synchronization pattern, we constructed a 2×4 table of category-level counts and applied a chi-square test of independence to determine whether the distribution of change activities differed significantly between platforms.

4.3.2.2 Exploring the lags (delays) of changes between platforms. To understand the prevalence of delays in communicating changes between GH and HF, we: (1) calculated the proportion of PTLM families exhibiting each type of lag (delay) by dividing the number of PTLM families in each lag category by the total number of families and multiplying by 100 to obtain percentages, which we then visualized using bar plots; (2) examined the prevalence of these lags across different project age groups by calculating the percentage of each lag type within each age category and visualizing the results using bar plots.

These findings offer insight into how promptly updates are coordinated between platforms. They help model developers and maintainers understand the distribution of synchronization practices—specifically, which platform tends to initiate activity. Although the analysis does not uncover the reasons behind these patterns, it highlights the prevalence and timing of update lags, providing a foundation for reflecting on current practices and identifying opportunities for better alignment in timing of release processes.

4.3.2.3 Exploring the synchronization patterns across PTLM family maturity. To understand how synchronization patterns are distributed and how they vary by PTLM family maturity, we applied the same calculation and visualization approach used in the lag analysis—this time focused on the eight synchronization patterns. Specifically, we examined the proportion of PTLM families exhibiting each pattern and how those proportions differ across age groups. These analyses help developers and maintainers reflect on how their synchronization practices evolve over time. Recognizing these patterns may support adjustments that lead to more consistent and timely synchronization between upstream (GH) and downstream (HF) activities as PTLMs mature.

4.3.2.4 Exploring the correlation between the number of PTLMs and contributor counts on synchronization patterns. Understanding whether coordination patterns correlate with the number of PTLM releases and contributors in a family can help reveal whether scale—both in terms of model proliferation and team size—affects how synchronization is managed across platforms. This insight can inform best practices for scaling multi-platform release workflows. To determine whether synchronization patterns correlate with the number of PTLM releases and contributors in each PTLM family, we:

1. Categorized each family's projects as "Single-release" (one PTLM) or "Multiple-release" (more than one PTLM) and created a symmetric bidirectional bar chart to visualize these distributions.
2. Grouped contributors by family and platform owner, counted unique contributors for each group, and used a boxplot to reveal platform-specific trends in synchronization strategies.
3. Applied the Spearman rank correlation coefficient—a nonparametric test suitable for evaluating monotonic relationships when data may not follow a normal distribution (Zar, 1972)—to evaluate the relationship between the number of contributors and the degree of temporal synchronization between platforms.

While synchronization patterns are originally nominal—representing categories without an inherent order—we converted them into ordinal variables to enable rank-based statistical analysis. This transformation was based on a proxy measure for temporal coordination intensity: the average number of overlapping biweekly activity periods between GH and HF within each PTLM family. To compute this, we first identified, for each PTLM family, the number of biweekly periods during which both platforms showed concurrent activity. We then calculated the average overlap count for all families that exhibited a given synchronization pattern. These averages allowed us to assign a ranking to each pattern, with higher overlap averages indicating stronger coordination.

Treating these ranked patterns as ordinal levels, we applied the Spearman rank correlation coefficient to assess whether the intensity of temporal coordination is associated with the number of unique contributors in a PTLM family. This analysis helps uncover whether larger teams are more likely to exhibit tightly synchronized release behaviors across platforms, which can inform best practices for managing model updates at scale.

4.3.2.5 Exploring the time taken for families across different lags (delays), synchronization patterns, and different maturity levels to communicate changes between GH and HF. To achieve this, we:

1. Measured the time it takes for a PTLM family to perform corresponding activities on the second platform after previously completing them on the first. This helps us understand the frequency and promptness of updates across platforms.
2. Examined whether these synchronization times significantly differ across various lags (delays) using the Kruskal-Wallis test—a nonparametric statistical test suitable for comparing three or more independent groups (GH-First, HF-First, Simultaneous in our case) when the data is not normally distributed (McKight and Najab, 2010). The Shapiro-Wilk test (Yazici and Yolacan, 2007) confirmed non-normality ($p = 0.00$), justifying the use of the Kruskal-Wallis test. When significant differences were observed ($p < 0.05$), we applied a post-hoc Dunn's test with Bonferroni correction[99] to identify pairwise group differences.
3. Calculated the average time differences (in days) between initial commit activities on one platform and the appearance of corresponding commits on the other within each synchronization pattern. This reflects the temporal synchronization behavior between platforms.
4. Tested whether these time intervals vary significantly across synchronization patterns using the same Kruskal–Wallis test approach, followed by Dunn-Bonferroni post-hoc tests where appropriate.
5. Assessed the average time taken by different maturity groups (based on PTLM family age) to coordinate changes across platforms to explore the effects of maturity on synchronization efficiency.
6. Used the Kruskal-Wallis test to evaluate differences in these synchronization times across maturity levels, followed by Dunn's test with Bonferroni correction when significant differences were found.

This multidimensional analysis offers insight into how synchronization timing varies by lags (delays), structural synchronization types, and project maturity, informing strategies for improving cross-platform synchronization in release engineering.

4.3.3 Results.

Most synchronization patterns (six out of eight) are dominated by model structure change type on GH, while HF emphasized external documentation, preprocessing, and model structure. The visualizations in Figure 18 and Figure 19 reinforce our earlier (global) findings of RQ1: GitHub exhibits a stable distribution of change types across synchronization patterns, confirming our earlier observations. In contrast, Hugging Face displays greater variation, with at least four patterns diverging noticeably from the platform-wide trend—suggesting a more pattern-sensitive approach to downstream model updates.

As shown in Figure 18, model structure leads in six synchronization patterns on GitHub—including Dense Partial (0.45), Intermittent (0.36), and Frequent Synchronization (0.32)—indicating a strong focus on defining and refining the model's architecture in the codebase. External documentation and training infrastructure also appear frequently among the top three, with external documentation playing a key role in patterns like Disperse and Sparse, and training infrastructure consistently ranking third across several others. The Rare Disjoint pattern stands out as the only case where external documentation (0.33) surpasses model structure, hinting at a shift toward improved clarity in less coordinated release settings.

On HF, external documentation is the leading change type across all synchronization patterns, ranging from 0.29 in Intermittent Synchronization to 0.56 in Frequent Synchronization. As illustrated in Figure 19, this dominance aligns with Hugging Face's downstream role in making models accessible to end users. Unlike GitHub, the secondary change types on Hugging Face are more varied. Preprocessing is frequently observed in Intermittent, Sparse, Frequent, and Rare Disjoint patterns, while output data and sharing are common in Disperse, Sparse, and Rare Disjoint. The Rare Synchronization pattern prioritizes pipeline performance (0.25) and

[99] https://help.easymedstat.com/support/solutions/articles/77000536997-dunn-bonferroni-post-hoc-test

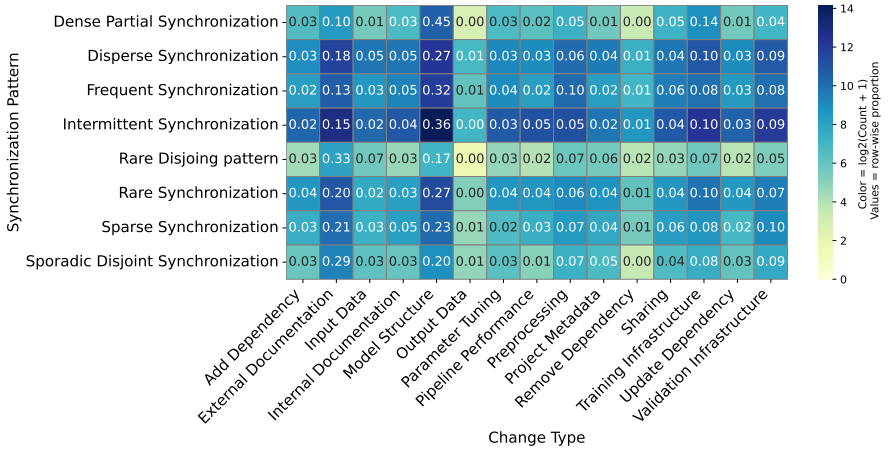

Fig. 18 Distribution of change types across synchronization patterns on GH. Cell color indicates $\log_2(\text{count} + 1)$ of commits per (pattern, label) pair, while cell text shows the row-wise proportion of each change type within a synchronization pattern, highlighting how different change types contribute to each pattern.

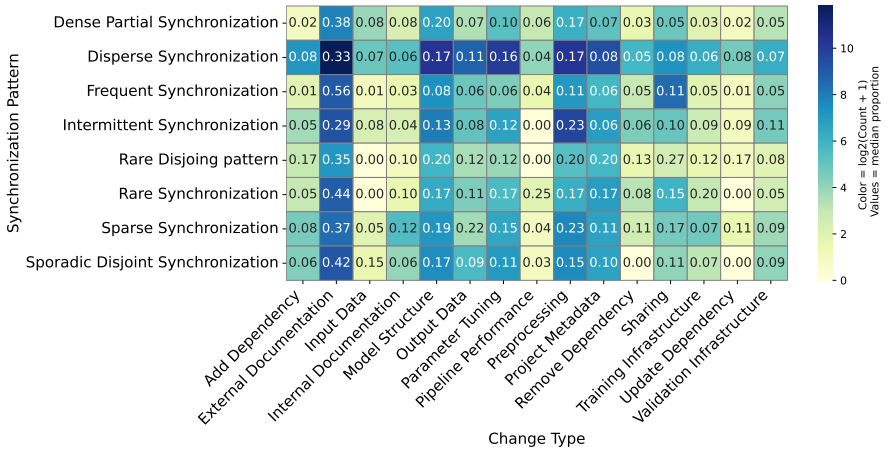

Fig. 19 Median proportion of change types across synchronization patterns on HF. Cell colour encodes $\log_2(\text{count} + 1)$ of raw change-type occurrences, while cell text reports the median proportion of each change type within the corresponding synchronization pattern.

training infrastructure (0.20), and Sporadic Disjoint highlights input data among its top changes—emphasizing the data-centric focus of downstream refinement.

To statistically assess whether platform-level differences in change type distributions are significant within each synchronization pattern, we constructed a 2 (platforms: GitHub, Hugging Face) × 4 (change categories) contingency table. All tests yielded highly significant p-values ($p < 0.05$), indicating that the distribution of change type categories differs meaningfully between GitHub and Hugging Face within each pattern. The strongest differences were observed in Frequent Synchronization ($\chi^2 = 698.33$, $p < 0.001$), followed by Disperse (411.25) and Rare (390.11) synchronization. Although the chi-square statistics were smaller for patterns like Sporadic Disjoint (44.26) and Rare Disjoint (13.27), they were still statistically significant, suggesting that even in less frequent or less coordinated release settings, platform-specific behavior—reflecting each platform's distinct focus on model development versus model sharing—diverges in meaningful ways.

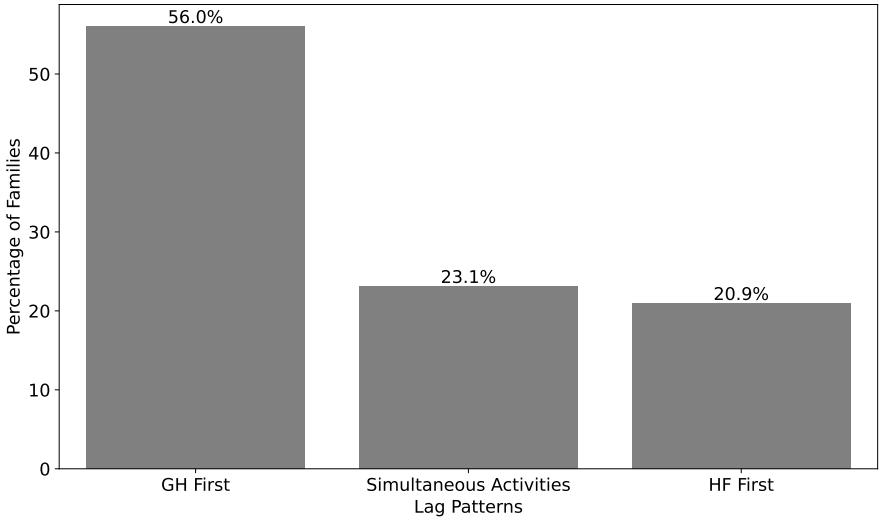

Fig. 20 The distribution of lags (delays) across the PTLM families on HF

Overall, these results indicate that the distinction between GitHub and Hugging Face holds consistently across all synchronization patterns. Regardless of how tightly or loosely coordinated the releases are, practitioners continue to use each platform in distinct ways, reinforcing a stable separation of roles in the release process.

Summary

The upstream–downstream divide is reflected in the dominance of structural changes on GH and documentation-focused changes on HF, with statistically significant differences in change type distributions across all synchronization patterns.

Most projects (56%) begin their commit activities on GH, compared to 20.9% that start on HF and 23.1% that initiate commits on both platforms within the same biweekly period. Figure 20 shows the distribution of lags (delays) across PTLM families. The distribution suggests that GH continues to serve as the primary platform for initiating development activities, especially for older projects that predate the rise of HuggingFace. For instance, projects like stanfordnlp/stanza and facebook/fasttext-language-identification began development on GH years before corresponding commit activities appeared on HF. This trend is consistent with the upstream–downstream workflow in software engineering, where core development happens upstream (on GH) and then flows downstream to distribution platforms like HF (Lin et al., 2022). This observation is further supported by our analysis of 39 Microsoft-hosted PTLMs, over 75% of which initiated development activities on GitHub. During an informal discussion at the ICSE 2025 conference, the first author learnt that model releases at Microsoft typically undergo multiple internal review cycles and cross-functional meetings before being published to HF. These coordination steps happen after development has already commenced on GitHub. This practical insight helps contextualize why commit activities are often observed first on GitHub—particularly among large organizations—before downstream propagation to HF.

Interestingly, the 23.1% of PTLM families with simultaneous activity on both platforms point to a shift in synchronization practices. Recent discussions on HuggingFace forums[100] show growing interest in synchronizing repositories across GH and HF, likely reflecting more mature, open-source-aware workflows that

[100] https://discuss.huggingface.co/t/github-repo-and-hugging-face-repo-sync/114697

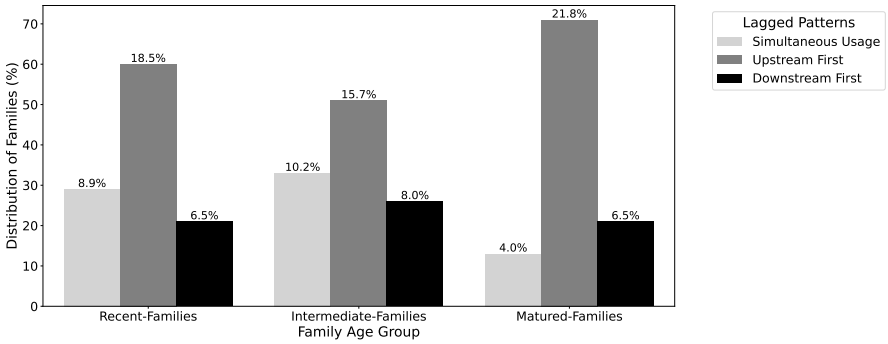

Fig. 21 The distribution of lagged patterns among HF PTLM families of different age groups, normalized by the number of families (%) in each group

prioritize synchronization from the outset. The 20.9% of families that begin on HuggingFace first may reflect a different strategy—one where practitioners prioritize early model availability, possibly releasing a pretrained model without immediately open-sourcing the accompanying training code. This aligns with observed practices in model distribution, where the focus is on accessibility and deployment readiness rather than immediate full-code transparency. Taken together, these findings highlight evolving norms in model release workflows. While a GH-first approach remains dominant—especially among older projects—more recent efforts are embracing tighter integration between GH and HF or even launching from HuggingFace, highlighting a growing awareness of platform synchronization and distribution needs.

Matured PTLM families contribute the most to the GH-first pattern observed earlier, with 21.8% starting their commit activities on GH. As shown in Figure 21, this pattern is also present in the recent (18.5%) and intermediate (15.7%) age groups, confirming that GH remains the most common starting point across all stages of project maturity.

The intermediate group shows the highest rate of simultaneous commit activities (10.2%), while this rate declines in the matured group (4.0%), suggesting that cross-platform synchronization is more common during mid-stage development. Surprisingly, recent projects do not lead in simultaneous activity. Although one might expect modern projects—given today's emphasis on openness and collaboration—to exhibit stronger synchronization, we observe the opposite trend. One potential explanation is that recent model developers might be training their models locally, using private GH repositories, and only publishing the model on HF once it reaches a stable version. They may later make the GH repository public. Since commit timestamps reflect the original creation dates rather than the public release dates, this development flow might manifest as a GH-first pattern in the data.

These findings suggest that synchronization practices evolve with project age but not in a strictly linear fashion. From a release engineering perspective, this highlights the need for workflows that accommodate phased releases and varying public visibility strategies across platforms.

Summary

Most projects (56%) begin their commit activities on GH—compared to 20.9% on HF and 23.1% initiating on both platforms within the same biweekly period—with matured PTLM families contributing most to the GH-first, accounting for 21.8% of GH-first.

***Disperse synchronization* (39.4%), *Sparse synchronization* (15.4%), and *Rare synchronization* (14.2%) are the most prevalent synchronization patterns observed in PTLM commit activities between GH and HF.** Figure 22 shows the distribution of synchronization patterns among PTLM families. The dominance of *Disperse synchronization* (39.4%) indicates partial synchronization, where some commit activities on HF align

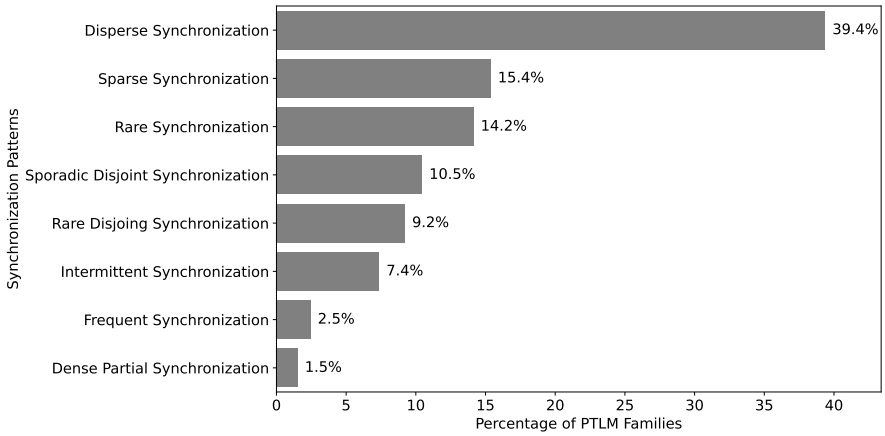

Fig. 22 The distribution of synchronization patterns among the HF PTLM families

with those on GH, while others occur independently. One possible interpretation is that synchronization between the two platforms is often handled in an ad hoc manner, raising concerns about users accessing outdated model versions. However, other factors could also contribute to this pattern, such as different release cadences for code and models, a deliberate separation of concerns between development and deployment teams, or platform-specific release strategies. *Sparse synchronization* (15.4%) reflects similar inconsistencies, marked by infrequent and loosely overlapping commit activities, while *Rare synchronization* (14.2%) shows limited but fully aligned updates that still lack regularity.

In contrast, *Dense partial synchronization* (1.5%) and *Frequent synchronization* (2.5%) are the least common patterns. The former reflects a shift from irregular to sustained synchronization (or vice versa), while the latter represents consistently coordinated commit activities across both platforms over an extended period. The rarity of these well-coordinated patterns underscores a broader challenge in release engineering. While better synchronization tools could simplify the process—allowing developers to coordinate updates across platforms in a single action—the low adoption of such patterns may also stem from limited awareness or undervaluation of cross-platform synchronization. Encouraging more automated workflows and educating practitioners on the benefits of consistent synchronization could enhance the accessibility and reliability of PTLMs for downstream users.

***Disperse synchronization* occurred most frequently in the intermediate and matured family age groups, while Rare Coordination was the most prevalent in the recent family age group.** Figure 23 reveals important shifts in synchronization behaviors as PTLM families mature. *Rare synchronization* and *Sparse synchronization* patterns decline significantly—*Rare synchronization* drops from 35.5% in recent families to just 1.0% in matured ones, while *Sparse synchronization* decreases from 22.7% to 5.5%. This reduction suggests that initial efforts to align commit activities across platforms are not sustained as projects grow. Meanwhile, less structured patterns like *Disperse synchronization* and *Sporadic Disjoint* show a steady rise—*Disperse synchronization* increases from 15.5% to 60%, and *Sporadic Disjoint* from 2.7% to 18.2%—indicating that commit activities become more fragmented over time. Since HF serves as the main access point for users, these patterns raise concerns about the timeliness and completeness of updates available to end-users.

Well-coordinated patterns such as *Frequent synchronization* and *Dense partial synchronization* are rare. *Frequent synchronization* occurs in 5.5% of intermediate families but drops to 1.8% in matured ones, and *Dense partial synchronization* appears in only 4.5% of intermediate families before vanishing entirely. This trend suggests that achieving synchronization is not only difficult to establish but even harder to maintain as complexity grows. This evolution may reflect shifting priorities or maintenance fatigue. These findings highlight a need to support not just the initiation of synchronization efforts but their sustainability—by reinforcing best practices early and integrating lightweight synchronization habits into standard development workflows,

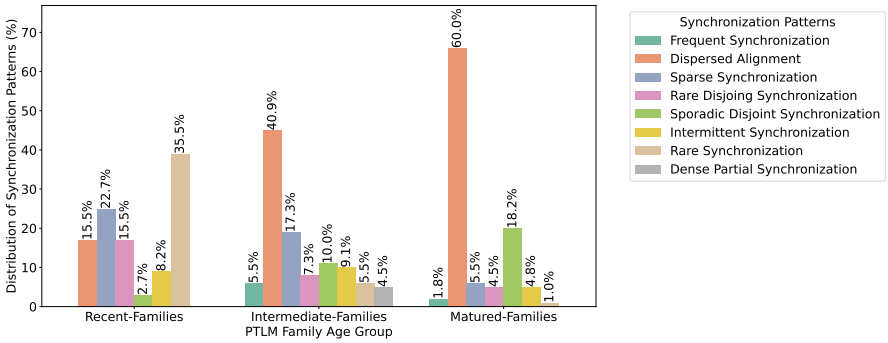

Fig. 23 The distribution of prevalent synchronization patterns in different maturity levels, where percentages add up to 100% for each pattern, normalized by the number of PTLM family (%) in each age group.

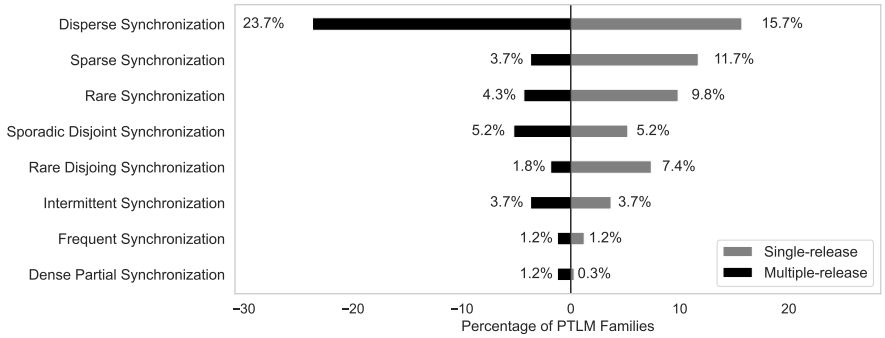

Fig. 24 Proportion of synchronization patterns in single-PTLM and multiple-PTLM families, with percentages adding up to 100% across both categories.

especially as projects move into long-term maintenance.

Summary

Disperse synchronization (39.4%), *Sparse synchronization* (15.4%), and *Rare synchronization* (14.2%) are the most prevalent synchronization patterns observed in PTLM commit activities across GH and HF, with *Disperse synchronization* occurring most frequently in the intermediate and matured family groups, and *Rare synchronization* dominating in the recent family age group.

The observed synchronization patterns are correlated with the number of model variants (PTLMs) within a PTLM family. Figure 24 shows the distribution of synchronization patterns across single-PTLM and multiple-PTLM families. Dispersed synchronization emerges as the most prevalent pattern, particularly in multiple-PTLM families (23.7%) compared to single-PTLM families (15.7%). This suggests that partial synchronization—where some activities across GH and HF overlap while others remain independent—is more common when multiple models are involved. While this may reflect the increased synchronization challenges in managing multiple variants simultaneously, it does not necessarily imply inefficiency; rather, it may indicate flexible release strategies tailored to each model's unique lifecycle.

Sparse synchronization is considerably more prominent in single-PTLM families (11.7%) than in multiple-PTLM families (3.7%), indicating that when only a single model is maintained, activities tend to occur in isolated bursts with extended periods of inactivity. Similarly, *Rare Disjoint* and *Rare synchronization patterns* are more common in single-PTLM families (7.4% and 9.8%, respectively) than in multiple-PTLM families

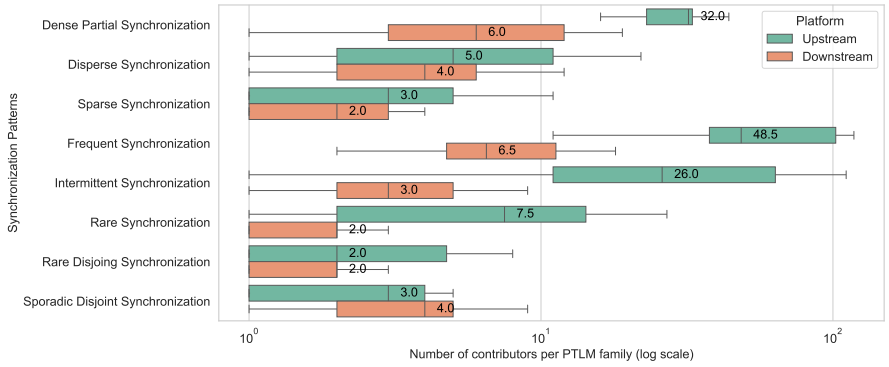

Fig. 25 Boxplot showing the distribution of the average (median) number of contributors per PTLM family for each synchronization pattern. The median value is annotated within each box for comparison.

(1.8% and 4.3%). These patterns reflect low-frequency activity that either occurs without synchronization or involves infrequent full synchronization between platforms.

Patterns such as *Sporadic Disjoint*, *Intermittent synchronization*, and *Frequent synchronization* display balanced distributions across both types of PTLM families, each appearing at low frequencies (ranging between 1.2% and 5.2%). *Dense partial synchronization*, characterized by a transition from sporadic to consistent synchronization, appears slightly more in multiple-PTLM families (1.2%) than in single-PTLM families (0.3%).

These findings highlight how the structural complexity of PTLM families shapes synchronization behavior. Overall, this suggests that the presence of multiple PTLM variants drives higher synchronization diversity and intensity, while single-variant families align more with intermittent or minimal synchronization behaviors.

synchronization patterns are correlated with the number of contributors across platforms, but a higher number of contributors does not always correspond to more structured synchronization. As shown in Figure 25, the contributor counts across synchronization patterns reflect distinct collaboration dynamics between GH and HF. In Rare, Intermittent, and *Frequent synchronization*, GH consistently has higher contributor counts (approximately 7.5, 26.0, and 48.5, respectively) than HF (around 2.0, 3.0, and 6.5). This supports the notion that a larger team on GH drives updates, with a smaller HF team mirroring these efforts at varying frequencies.

In Dispersed synchronization, contributor counts are more balanced (5.0 GH vs. 4.0 HF), indicating partial synchronization where contributions are occasionally coordinated but also occur independently. Similarly, *Sparse synchronization* involves low contributor counts on both platforms (3.0 and 2.0), consistent with infrequent, minimal collaborative activity. *Dense partial synchronization* shows a notable difference (32.0 GH vs. 6.0 HF), suggesting a two-phase process where GH initiates and sustains activity, while HF plays a more active role in the latter, coordinated phase. In *Sporadic Disjoint* and *Rare Disjoint patterns*, both platforms have low contributor counts (around 3.0 and 4.0 for *Sporadic Disjoint*; 2.0 for *Rare Disjoint*), reflecting disjointed efforts with minimal interaction.

Overall, contributor distribution highlights how GH often leads model development with larger teams, especially in more dynamic synchronization patterns, while HF either supports or mirrors those efforts to varying degrees depending on the synchronization structure.

To further probe this observation, we computed the Spearman correlation between synchronization patterns and contributor counts. Interestingly, the results reveal a strong, statistically significant negative correlation on both platforms (HF: -0.8743, $p < 0.001$; GH: -0.9701, $p < 0.001$), suggesting that as the number of contributors increases, projects are more likely to exhibit dispersed or loosely coordinated commit behaviors. This implies that larger teams may contribute to more fragmented synchronization patterns, possibly due to parallel development streams, asynchronous work schedules, or decentralized release responsibilities—challenging the assumption that more contributors naturally lead to more tightly coordinated release cycles.

Table 3: Statistical test of average time taken by each PTLM family to reflect changes between the two platforms in each synchronization pattern. DPS: Dense partial synchronization, DS: Dispersed synchronization, FS: Frequent synchronization, IS: Intermittent synchronization, RS: Rare synchronization, RD: Rare Disjoint, SS: Sparse synchronization, SD: Sporadic Disjoint

	DPS	**DS**	**FS**	**IS**	**RD**	**RS**	**SS**	**SD**
DPS	–							
DS	<	–						
FS	<	<	–					
IS	<	<	x	–				
RD	x	<	<	x	–			
RS	<	<	<	<	<	–		
SS	<	x	<	<	<	<	–	
SD	<	<	<	<	<	x	<	–

Summary

synchronization patterns in PTLM families are shaped by both the number of model variants (PTLMs) and contributors, with multiple-PTLM families exhibiting more diverse and intense synchronization behaviors, while an increase in contributors—especially downstream—correlates with more fragmented and less structured synchronization.

The families that start their development activities on HF spend more days before performing any form of activities on GH, while those using the two platforms simultaneously switch between them more quickly. Although our previous findings indicate that some PTLM families use both platforms simultaneously, we did not find evidence that these families employ continuous integration to automatically coordinate commits between GH and HF. Here, continuous integration refers to the automated, real-time syncing of changes made on one platform with the other. Instead, the average delay of 9.32 days between activities on the two platforms suggests that synchronization is likely managed manually or through non-automated workflows.

The results show that PTLM families beginning on HF take an average of 22.35 days to initiate activities on GH, while those starting on GH take an average of 15.82 days to begin on HF.

A Kruskal-Wallis test revealed that the differences in average synchronization time among the three groups—GH First, HF First, and Simultaneous—are statistically significant. Specifically, the ranking of synchronization delays from highest to lowest is: HF First, GH First, and Simultaneous, with all pairwise differences being significant. This suggests that the pattern of platform engagement is associated with variation in the speed of synchronization between GH and HF.

PTLM families with *Frequent* and *Intermittent synchronization* engage in activities on the other platform much sooner, while families following *Rare Disjoint* and *Sporadic Disjoint* patterns experience much longer delays. The time before PTLM families conduct activities on the second platform varies significantly across synchronization patterns, with well-coordinated patterns like *Frequent synchronization* and *Intermittent synchronization* showing quicker engagement between platforms. *Frequent synchronization* families take the shortest time (0.87 days), followed by *Intermittent synchronization* (1.17 days) and *Dense partial synchronization* (3.68 days). These quicker transitions could be due to automated synchronization tools or dedicated teams handling cross-platform updates efficiently. In contrast, patterns such as *Sparse synchronization* (26.98 days), *Rare Disjoint* (109.39 days), and *Sporadic Disjoint* (127.39 days) show much longer delays, likely due to the lack of structured synchronization processes, which result in less timely updates between platforms and less engagement from the project team on the second platform.

Furthermore, Table 3 shows the statistical significance of the differences in the time spent by PTLM families in each synchronization pattern to reflect commit change types between the two platforms. We used Dunn's post-hoc test with Bonferroni correction to account for multiple comparisons. In the table, "<" denotes that the Bonferroni-corrected p-value is less than 0.05, while "x" indicates comparisons where the corrected p-value

was greater than 0.05 and thus not statistically significant. Overall, statistical tests indicate that most synchronization patterns show significant differences in the time taken to perform activities on the other platform. However, some comparisons did not yield statistically significant differences. Specifically, the differences between *Frequent synchronization* and *Intermittent synchronization*, *Frequent synchronization* and *Rare synchronization*, *Intermittent synchronization* and *Rare synchronization*, *Dense partial synchronization* and *Dispersed synchronization*, *Dense partial synchronization* and *Rare synchronization*, and *Rare Disjoint* and *Sporadic Disjoint* were not statistically significant.

The time to communicate changes between GH and HF increases progressively with project maturity, with matured families experiencing significantly longer delays (24 days). Our findings show that PTLM family in the recent and intermediate age groups spend, on average, 9 days to communicate changes between platforms, while families in the matured age group take significantly longer, averaging 24 days. These differences suggest that as PTLM families mature, the time spent in coordinating activities across platforms increases. A Kruskal-Wallis test was conducted to compare the time taken to communicate changes across the three age groups. The results reveal significant differences, with p-values as follows: Intermediate vs. Matured ($p < 0.05$), Intermediate vs. Recent ($p < 0.05$), and Matured vs. Recent ($p < 0.05$). These findings indicate that PTLM families in the recent age group coordinate commit change types more quickly than both matured and intermediate families, with matured families experiencing the longest delays in cross-platform communication.

Summary

PTLMs using both platforms experience an average synchronization delay of over 9 days, with *Frequent synchronization* families updating the fastest (0.87 days) and *Sparse synchronization*, *Rare Disjoint*, and *Sporadic Disjoint* families showing increasingly longer delays (26.98, 109.39, and 127.39 days, respectively), indicating significant variation in update times and revealing that mature projects tend to have more stable synchronization and slower cross-platform communication.

5 Discussion and Implication

5.1 Discussion

The increasing availability of PTLMs has sparked growing interest in model repositories and release engineering practices (Min et al., 2023). HF, as one of the largest hosting platforms for these models (Jiang et al., 2023b), plays a central role in shaping their release processes. We argue that these repositories represent only one link in the broader supply chain of AI model development. Our study focuses on the synchronization between model repositories on HF and their upstream code repositories on GH. By examining the types of changes in the commit activities, how these commit are synchronized across these platforms, and the dynamics of these synchronization patterns, we offer a comprehensive view of cross-repository PTLM development. This analysis is grounded in 904 PTLMs linked to 325 GH repositories, encompassing 140,000 GH commits and 17,000 HF commits.

To contextualize our analysis, we begin by distinguishing between synchronization and coordination to clarify cross-platform activity between GH and HF. Synchronization refers to the timing of commits across platforms, while coordination also involves managing interdependencies, aligning goals, and maintaining shared understanding among contributors. For PTLMs, coordination is difficult to assess directly due to the complexity of their supply chain, which spans datasets, model weights, configurations, licenses, and distributed contributors. These layers introduce coordination challenges that are not easily visible through code alone. As a result, we focus on synchronization as a practical first step.

Building on this framing, our findings confirm a clear division of labor between GH and HF, consistent with their roles in the PTLM release workflow. PTLM families tend to exhibit moderate-to-high similarity in

change types across platforms, suggesting some level of synchronization. At the high level of change taxonomy, the most prevalent types on GH are model structure, external documentation, and training infrastructure, while HF prioritizes external documentation, model structure, and preprocessing. Despite this apparent overlap—particularly in model structure and documentation—our topic-level analysis reveals that each platform emphasizes different facets of these shared categories. On GH, model structure changes are primarily code-driven, involving large-scale refactoring (BigRefactor), task-specific updates (MultipleChoiceTask), and restructuring of training scripts and infrastructure (e.g., Makefile, Directory). In contrast, HF's model structure topics focus on uploading model weights, specifying architecture components, and integrating models into the HF ecosystem. A similar divergence appears in external documentation: GH emphasizes community-facing documentation such as structural layouts and feature explanations, while HF focuses on formal artifacts like licensing terms, research citations, and fine-tuning prompt templates. These differences indicate that even when PTLM updates fall under the same broad categories, the platforms serve distinct functions within the release pipeline. However, synchronization across these roles remains partial and temporally irregular, as indicated by the prevalence of the *Disperse synchronization pattern* found in RQ3.

These platform distinctions are further influenced by the stage of PTLM family maturity. In the early stages of PTLM families, commit activities are often concentrated on a single platform—typically GitHub—while activity on Hugging Face is minimal. When updates do occur on both platforms at this stage, they tend to be tightly aligned in time, resulting in *Rare synchronization patterns* that are minimal but fully synchronized. As projects mature, the volume of activity across both platforms increases, but these updates become more fragmented, with only a subset of changes occurring in close temporal proximity. This leads to the prevalence of *Disperse synchronization* in intermediate and mature PTLM families, where synchronization exists but is partial and inconsistently timed. This trend may reflect challenges similar to those observed in large-scale software systems, where misaligned planning practices—such as fragmented specification, prioritization, estimation, and allocation—hinder inter-team coordination and lead to delays or redundant work (Bick et al., 2017). Accordingly, the average time lag between cross-platform updates increases with maturity, reaching 109 days in a case like *Rare Disjoint Synchronization*. Because our dataset focuses on models with over 10,000 downloads, it likely captures PTLMs that are already in their intermediate or mature stages, as more than 50% of the PTLMs in our dataset are over two years old. This may bias our observations toward Disperse synchronization patterns—where activities often begin on one platform, particularly GH, before gradually extending to HF. This trend may reflect that many mature models were originally maintained on GH before later adopting HF for broader distribution.

In addition to platform roles and maturity, contributor behavior adds another layer of complexity to synchronization dynamics across platforms. While it might be intuitive to assume that larger contributor bases facilitate smoother synchronization, our findings indicate the opposite: PTLM families with more contributors—particularly on HF—tend to exhibit more fragmented synchronization patterns. Although PTLM families with high change similarity scores also show a greater proportion of multi-author contributions across GH and HF, this does not consistently translate into synchronized activity. This apparent contradiction reflects known challenges in large-scale distributed collaboration. Prior research has shown that, despite the benefits of developer coordination—such as improved software quality—teams composed of diverse and widely distributed contributors often face obstacles such as time zone differences, strategic misalignment, limited knowledge sharing, geographical distance, awareness gaps, and organizational boundaries (Suali et al., 2017). These factors may interrupt the continuity of updates across platforms, leading to disjointed contributions even when collaborative intent exists. In the context of PTLMs—where contributors are drawn from research institutions, industry, and open-source communities—these barriers likely contribute to the prevalent *Disperse synchronization pattern* observed in our study.

5.2 How the observed synchronization patterns relate to versioning practices

Although our study does not directly analyze model versioning, the observed synchronization patterns between GH and HF—particularly *Disperse*, *Sparse*, and *Rare* Synchronization—raise practical concerns about the consistency and traceability of PTLM versions across platforms. For instance, in the case of the aubmindlab/bert-

base-arabert model, the initial GitHub repository [101] was created on February 21, 2020, while the corresponding Hugging Face model page was established shortly afterward, on February 27, 2020. Notably, on March 12, 2020, a critical update was made on GitHub that corrected the model training procedure by changing the training step parameter from a hardcoded *TRAIN_STEPS* to the variable *num_train_steps*. This adjustment, while syntactically minor, could significantly alter the resulting model's behavior: if the number of training steps changed, the model might become either undertrained or overfitted, leading to degraded or altered performance.

However, Hugging Face did not reflect any corresponding update in the model weights or documentation until much later—specifically, on July 7, 2020, with an update to the *tf_model.h5* file. Even then, there is no clear indication that this new file incorporates the March 12 fix, nor is there any metadata explicitly connecting that model weight to the earlier upstream change. This disconnect exemplifies the risks posed by loosely coordinated synchronization: downstream users accessing the Hugging Face model in the interim would have relied on potentially outdated or suboptimal weights, without any visibility into important upstream corrections.

Our findings, coupled with this example, emphasize a broader challenge: when synchronization between upstream and downstream platforms is irregular or delayed, it becomes exceedingly difficult to track what constitutes a new "version" of a model. Without a semantic versioning scheme or provenance metadata—such as change logs, compatibility information, or training parameters—users cannot reliably determine whether they are using a model version that reflects critical updates. As argued in our prior work (Ajibode et al., 2025), the multidimensional nature of PTLMs (e.g., changes to architecture, data, or training procedure) makes one-dimensional version numbers inadequate. Thus, better synchronization practices must go hand in hand with versioning standards to ensure reproducibility, user trust, and correct downstream adoption.

5.3 Implications

Our findings offer practical implications for different stakeholders involved in the development and maintenance of PTLMs across platforms.

Producers (Upstream pre-trained model developers and contributors):

- Synchronization between GitHub and HF is often delayed—especially in mature projects—suggesting that contributors should proactively structure releases and assign responsibilities across platforms.
- Our findings show that large contributor teams do not necessarily ensure synchronization, highlighting the need for clearer role assignments and maintenance strategies.
- Synchronizing key updates such as documentation and variant releases across platforms may help reduce drift and improve the consistency of model families.

Consumers (Downstream users of pre-trained models):

- Users should be cautious of assuming that both platforms reflect the latest model state. Delays and fragmentation are common, which may result in outdated models or documentation on one platform.
- Since GitHub emphasizes technical implementation and HF focuses on user-facing artifacts, users should consult both to fully understand a model's current functionality and limitations.

Researchers:

- Future work can explore how synchronization affects coordination outcomes—such as successful propagation of changes or functional consistency across platforms.
- There is also an opportunity to examine which social or technical mechanisms (e.g., contributor identity, tooling, communication norms) promote or hinder effective coordination between upstream and downstream platforms.

[101] https://github.com/aub-mind/arabert

6 Threats to Validity

6.1 Internal Validity

A potential threat to internal validity stems from our decision to apply a 10,000-download threshold when selecting models. While this filter helped us focus on high-quality, widely adopted PTLMs, it may have resulted in the underrepresentation of less popular or emerging models—especially those that share a GitHub repository with the included models but fall below the threshold. This incompleteness in capturing full PTLM families could obscure differences in release engineering behaviors across a wider spectrum of model popularity and maintenance practices. Nonetheless, we chose this threshold to improve the reliability of our analysis by ensuring adequate development activity and documentation. More importantly, this decision was grounded in the need for thorough manual analysis across all research questions. Specifically, we manually verified and curated GitHub links for 971 repositories, labeled 1,600 commits for RQ1, and reviewed 177 synchronization patterns for RQ2. Future studies may revisit this trade-off to capture broader PTLM families, including those with lower visibility or niche use cases.

Another threat comes from the automated filtering of GH links. By retaining only links containing either the owner's name or model name, we aimed to reduce noise and ensure the dataset's relevance. However, this approach may have inadvertently excluded models with unconventional repository naming structures, potentially limiting the completeness of the dataset.

Additionally, the selection of the top 10 most significant change topics for each change type posed a potential threat. In cases where fewer than 10 topics were available—which occurred only on HF—we selected all the available topics. When no topic was available, also only on HF, we used "No identified instances" as a placeholder and proceeded with the analysis using only GH data. While this ensured consistency in the number of topics considered across platforms, it may have limited the representation of specific types of model maintenance or release activities on HF, such as bug fixes or weight uploads. However, this limitation did not affect the core results of our comparison, as our approach maintained alignment in topic coverage between the two platforms.

Finally, the choice of a bi-weekly interval for aggregating commit activities may have affected the granularity of the analysis. While this interval balanced capturing trends and avoiding clutter, it could misclassify commit patterns in projects with more frequent or slower updates. For instance, a project with concentrated commits in a short time span might be misclassified as "rare" or "infrequent." Nevertheless, the bi-weekly window was necessary to capture meaningful synchronization events without overwhelming the visualization, especially for long-lived projects. Despite its limitations, this approach was essential for clear and interpretable results.

6.2 External Validity

A key threat to external validity is the focus on NLP models, which may limit the generalizability of our findings to other domains such as Computer Vision or Reinforcement Learning. Different development practices, repository structures, and documentation standards across these domains could lead to distinct synchronization patterns. While the three synchronization characteristics we examined—lag (delay), synchronization types, and intensity—are likely to be present in other domains, different combinations or interactions of these characteristics may arise. For example, a domain might exhibit high-intensity synchronization with minimal lag but still follow a distinct synchronization type not observed in NLP. Since our methodology is tailored to NLP models, its direct application to other domains may require adjustments to account for domain-specific practices. Nevertheless, our approach offers a structured framework that can be adapted for future studies on model release and synchronization in other fields.

6.3 Construct Validity

A potential limitation arises from the use of a large language model (LLM) for commit labeling, as the model may introduce inherent biases in its outputs. While we mitigated random variations by repeating the labeling process 10 times per platform and averaging the results, this approach does not eliminate potential systematic biases inherent in the LLM's pretraining data. To improve the reliability of the analysis, a substantial portion of the labeled data was manually reviewed and validated by the first and second authors. Consequently, some classifications may still reflect the model's biases rather than the true nature of the commits. Future work could address this limitation more systematically by incorporating human-in-the-loop validation for bias detection and correction.

Another limitation is our use of the taxonomy from Bhatia et al. (2023), originally designed for general ML applications, which may not fully capture PTLM-specific change types—such as tokenizer updates, adapter integration, or fine-tuning data format changes. While this framework served as a useful starting point and has been applied in similar contexts (e.g., (Castaño et al., 2024a)), we acknowledge that some relevant changes unique to LLM development may be misclassified or overlooked. Future work could refine this taxonomy to better reflect PTLM-specific development activities.

7 Conclusion

This study investigated the synchronization of commit activities between GH and HF platforms for PTLMs, using a mixed-methods approach. We addressed three main research questions: the types of commit activity changes across platforms, synchronization patterns, and the dynamics of these synchronization patterns across PTLM families.

Our findings show that while GH commits tend to emphasize training infrastructure, model structure, and external documentation, HF activities are more focused on preprocessing steps, external documentation, and model structure. While 77% of projects show moderate to high similarity in commit activity across platforms, this distribution of changes aligns with the roles these platforms play—GH often driving upstream development and infrastructure setup, while HF supports downstream adaptation and usage refinement.

We identified eight synchronization patterns, with the most common being *Dispersed synchronization* (39.4%), where changes take an average of 19 days to propagate between platforms. The synchronization pattern evolves as projects mature, with newer projects exhibiting *Rare synchronization* and more mature ones shifting toward Dispersed synchronization. Our analysis indicates that while more contributors can increase activity, it does not guarantee better synchronization, especially as PTLM families mature and contributor engagement declines.

These insights highlight the need for better synchronization and approaches that support synchronization, particularly for PTLM families at different maturity levels.

Data Availability

The datasets generated and analyzed during this study are available in the replication package (Adekunle, 2025).

Funding

This research was supported by the NSERC Discovery Grant RGPIN-2025-04654.

Ethical Approval

This study does not involve human participants or animals.

Informed Consent

No human subjects were involved in this study.

Conflicts of Interests/Competing Interests

The authors declare that they have no known competing interests or personal relationships that could have (appeared to) influenced the work reported in this article.

Author Contributions

- Adekunle Ajibode: Conceptualization, Data Collection, Methodology, Data Analysis, Writing – Original Draft.
- Abdul Ali Bangash: Methodology, Data Validation, Writing – Review & Editing.
- Bram Adams: Supervision, Writing – Review & Editing, Conceptual Guidance, Research Direction.
- Ahmed E. Hassan: Supervision, Research Direction.

References

Ajibode Adekunle. paper_2, 2025. URL `https://github.com/eyinlojuoluwa/paper_2`. GitHub repository.

Bridgwater Adrian. What is upstream/downstream software? `https://www.computerweekly.com/blog/Open-Source-Insider/What-is-upstream-downstream-software`, 2016. Accessed: 2025-04-03.

Adem Ait, Javier Luis Cánovas Izquierdo, and Jordi Cabot. On the suitability of hugging face hub for empirical studies. *Empirical Software Engineering*, 30(2):1–48, 2025.

Adekunle Ajibode, Abdul Ali Bangash, Filipe R Cogo, Bram Adams, and Ahmed E Hassan. Towards semantic versioning of open pre-trained language model releases on hugging face. *Empirical Software Engineering*, 30(3):1–63, 2025.

Haldun Akoglu. User's guide to correlation coefficients. *Turkish journal of emergency medicine*, 18(3):91–93, 2018.

Philippe Aubry, Gwenaël Quaintenne, Jeremy Dupuy, Charlotte Francesiaz, Matthieu Guillemain, and Alain Caizergues. On using stratified two-stage sampling for large-scale multispecies surveys. *Ecological Informatics*, 77:102229, 2023.

Marthe Berntzen, Viktoria Stray, and Nils Brede Moe. Coordination strategies: managing inter-team coordination challenges in large-scale agile. In *International Conference on Agile Software Development*, pages 140–156. Springer, 2021.

Aaditya Bhatia, Ellis E Eghan, Manel Grichi, William G Cavanagh, Zhen Ming Jiang, and Bram Adams. Towards a change taxonomy for machine learning pipelines: Empirical study of ml pipelines and forks related to academic publications. *Empirical Software Engineering*, 28(3):60, 2023.

Saskia Bick, Kai Spohrer, Rashina Hoda, Alexander Scheerer, and Armin Heinzl. Coordination challenges in large-scale software development: a case study of planning misalignment in hybrid settings. *IEEE Transactions on Software Engineering*, 44(10):932–950, 2017.

Thomas Bock, Claus Hunsen, Mitchell Joblin, and Sven Apel. Synchronous development in open-source projects: A higher-level perspective. *Automated Software Engineering*, 29(1):3, 2022.

Joel Castaño, Rafael Cabañas, Antonio Salmerón, David Lo, and Silverio Martínez-Fernández. How do machine learning models change? *arXiv preprint arXiv:2411.09645*, 2024a.

Joel Castaño, Silverio Martínez-Fernández, Xavier Franch, and Justus Bogner. Analyzing the evolution and maintenance of ml models on hugging face. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 607–618. IEEE, 2024b.

Eric Chagnon, Ronald Pandolfi, Jeffrey Donatelli, and Daniela Ushizima. Benchmarking topic models on scientific articles using berteley. *Natural Language Processing Journal*, 6:100044, 2024.

Kim Cocks and David J Torgerson. Sample size calculations for pilot randomized trials: a confidence interval approach. *Journal of clinical epidemiology*, 66(2):197–201, 2013.

Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work*, pages 1277–1286, 2012.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Themistoklis Diamantopoulos, Dimitrios-Nikitas Nastos, and Andreas Symeonidis. Semantically-enriched jira issue tracking data. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 218–222. IEEE, 2023.

Khaled El Emam. *Benchmarking kappa for software process assessment reliability studies*. Fraunhofer-IESE, 1998.

Armstrong Foundjem and Bram Adams. Release synchronization in software ecosystems: Empirical study on openstack. *Empirical Software Engineering*, 26:1–50, 2021.

Rosalba Giuffrida and Yvonne Dittrich. A conceptual framework to study the role of communication through social software for coordination in globally-distributed software teams. *Information and Software Technology*, 63:11–30, 2015.

Google. Gemini api pricing, 2024. URL `https://ai.google.dev/pricing`. Accessed on December 9, 2024.

Maarten Grootendorst. Bertopic: Neural topic modeling with a class-based tf-idf procedure. *arXiv preprint arXiv:2203.05794*, 2022.

Haiqiao Gu, Hao He, and Minghui Zhou. Self-admitted library migrations in java, javascript, and python packaging ecosystems: A comparative study. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 627–638. IEEE, 2023.

James D Herbsleb, Audris Mockus, Thomas A Finholt, and Rebecca E Grinter. An empirical study of global software development: distance and speed. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 81–90. IEEE, 2001.

Tjaša Heričko and Boštjan Šumak. Commit classification into software maintenance activities: A systematic literature review. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1646–1651. IEEE, 2023.

Abram Hindle, Daniel M German, and Ric Holt. What do large commits tell us? a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108, 2008.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

GI Ivchenko and SA Honov. On the jaccard similarity test. *Journal of Mathematical Sciences*, 88:789–794, 1998.

Mario Janke and Patrick Mäder. 7 dimensions of software change patterns. *Scientific Reports*, 14(1):6141, 2024.

Wenxin Jiang, Jason Jones, Jerin Yasmin, Nicholas Synovic, Rajeev Sashti, Sophie Chen, George K Thiruvathukal, Yuan Tian, and James C Davis. Peatmoss: Mining pre-trained models in open-source software. *arXiv preprint arXiv:2310.03620*, 2023a.

Wenxin Jiang, Nicholas Synovic, Matt Hyatt, Taylor R Schorlemmer, Rohan Sethi, Yung-Hsiang Lu, George K Thiruvathukal, and James C Davis. An empirical study of pre-trained model reuse in the hugging face deep learning model registry. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*,

pages 2463–2475. IEEE, 2023b.

Geetha Kanaparan and Diane E Strode. Investigating the relationship between coordination strategy and coordination effectiveness in agile software development projects. *Information and Software Technology*, 182: 107708, 2025.

Alexander Krause-Glau, Marcel Bader, and Wilhelm Hasselbring. Collaborative software visualization for program comprehension. In *2022 Working Conference on Software Visualization (VISSOFT)*, pages 75–86. IEEE, 2022.

Robert E Kraut and Lynn A Streeter. Coordination in software development. *Communications of the ACM*, 38 (3):69–81, 1995.

Ye Li and Alexander Maedche. Formulating effective coordination strategies in agile global software development teams. 2012.

Jiahuei Lin, Haoxiang Zhang, Bram Adams, and Ahmed E Hassan. Upstream bug management in linux distributions: An empirical study of debian and fedora practices. *Empirical Software Engineering*, 27(6):134, 2022.

Sihai Lin, Yutao Ma, and Jianxun Chen. Empirical evidence on developer's commit activity for open-source software projects. In *Seke*, volume 13, pages 455–460, 2013.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.", 2012.

Thomas Magelinski, Lynnette Ng, and Kathleen Carley. A synchronized action framework for detection of coordination on social media. *Journal of Online Trust and Safety*, 1(2), 2022.

Thomas W Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys (CSUR)*, 26(1):87–119, 1994.

Potsawee Manakul, Adian Liusie, and Mark JF Gales. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. *arXiv preprint arXiv:2303.08896*, 2023.

Huanru Henry Mao. A survey on self-supervised pre-training for sequential transfer learning in neural networks. *arXiv preprint arXiv:2007.00800*, 2020.

Mary L McHugh. The chi-square test of independence. *Biochemia medica*, 23(2):143–149, 2013.

Patrick E McKight and Julius Najab. Kruskal-wallis test. *The corsini encyclopedia of psychology*, pages 1–1, 2010.

Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. Recent advances in natural language processing via large pre-trained language models: A survey. *ACM Computing Surveys*, 56(2):1–40, 2023.

Mockus and Votta. Identifying reasons for software changes using historic databases. In *Proceedings 2000 international conference on software maintenance*, pages 120–130. IEEE, 2000.

R OpenAI. Gpt-4 technical report. arxiv 2303.08774. *View in Article*, 2:13, 2023.

Jorge Pérez, Jessica Díaz, Javier Garcia-Martin, and Bernardo Tabuenca. Systematic literature reviews in software engineering—enhancement of the study selection process using cohen's kappa statistic. *Journal of Systems and Software*, 168:110657, 2020.

Ajay S Singh and Micah B Masuku. Sampling techniques & determination of sample size in applied statistics research: An overview. *International Journal of economics, commerce and management*, 2(11):1–22, 2014.

Trevor Stalnaker, Nathan Wintersgill, Oscar Chaparro, Laura A Heymann, Massimiliano Di Penta, Daniel M German, and Denys Poshyvanyk. The ml supply chain in the era of software 2.0: Lessons learned from hugging face. *arXiv preprint arXiv:2502.04484*, 2025.

Viktoria Stray and Nils Brede Moe. Understanding coordination in global software engineering: A mixed-methods study on the use of meetings and slack. *Journal of Systems and Software*, 170:110717, 2020.

Diane E Strode and Sid L Huff. A coordination perspective on agile software development. In *Modern Techniques for Successful IT Project Management*, pages 64–96. IGI Global Scientific Publishing, 2015.

Diane E Strode, Sid L Huff, Beverley Hope, and Sebastian Link. Coordination in co-located agile software development projects. *Journal of Systems and Software*, 85(6):1222–1238, 2012.

AJ Suali, SSM Fauzi, WAWM Sobri, and MHNM Nasir. Developers' coordination issues and its impact on software quality: A systematic review. In *2017 3rd International Conference on Science in Information Technology (ICSITech)*, pages 659–663. IEEE, 2017.

ABM Talukder, Mali Senapathi, and Jim Buchan. Coordination in distributed agile software development: a systematic review. 2017.

Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. What makes a good commit message? In *Proceedings of the 44th International Conference on Software Engineering*, pages 2389–2401, 2022.

Susana M Vieira, Uzay Kaymak, and João MC Sousa. Cohen's kappa coefficient as a performance measure for feature selection. In *International conference on fuzzy systems*, pages 1–8. IEEE, 2010.

Haifeng Wang, Jiwei Li, Hua Wu, Eduard Hovy, and Yu Sun. Pre-trained language models and their applications. *Engineering*, 2022.

Shenao Wang, Yanjie Zhao, Xinyi Hou, and Haoyu Wang. Large language model supply chain: A research agenda. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–46, 2025.

Jed R Wood and Larry E Wood. Card sorting: current practices and beyond. *Journal of Usability Studies*, 4(1): 1–6, 2008.

Meng Yan, Ying Fu, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D Kymer. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *Journal of Systems and Software*, 113:296–308, 2016.

Berna Yazici and Senay Yolacan. A comparison of various tests of normality. *Journal of statistical computation and simulation*, 77(2):175–183, 2007.

Jerrold H Zar. Significance testing of the spearman rank correlation coefficient. *Journal of the American Statistical Association*, 67(339):578–580, 1972.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

Zhimin Zhao, Yihao Chen, Abdul Ali Bangash, Bram Adams, and Ahmed E Hassan. An empirical study of challenges in machine learning asset management. *Empirical Software Engineering*, 29(4):98, 2024.

Appendix A: Prompt structure used for the commit activities classification of PTLM families.

```
You are a helpful assistant tasked with classifying commit messages.
Classify each commit message with one of the below classifications and return the
results as a JSON array of objects, where each object has two keys: "commit" (the
commit message) and "label" (the classification label).
Note: Don't skip any commit without labeling, even if there are multiple
occurrences.

Categories:
- External Documentation: Changes to end-user documentation (e.g., create
    readme.md, first draft of model card)
- Internal Documentation: Changes explaining code workings internally (e.g.,
    update merges.txt)
- Model Structure: Structural changes to the model's code (e.g., bug fix, fix
    eos_token and bos_token in config (#1), add model 1.3.1, add new
    sentencetransformer model, add tf weights)
- Training Infrastructure: Changes affecting the model training logic (e.g.,
    iter_0600000, training infrastructure)
- Preprocessing: Changes related to data manipulation before it reaches model
    training (e.g., add missing files and fix tokenizer_config.json (#6), add
    preprocessing file, add tokenizer)
- Parameter Tuning: Adjustments to hardcoded hyperparameters within the ML
    pipeline (e.g., update config for hf transformers)
- Pipeline Performance: Modifications enhancing ML run-time pipeline efficiency
    (e.g., fix rows mixing)
- Validation Infrastructure: Modifications to components evaluating model
    performance (e.g., adding evaluation results (#7), update benchmark
    comparisons to add openchat and jackalope, update benchmark scores and
    averages for 1.1)
- Input Data: Changes to logic for loading or ingesting external data (e.g., new
    data, correct pre-training data)
- Output Data: Modifications to how output data is stored
- Sharing: Changes that enable better collaboration (e.g., from
    git://github.com/01-ai/yi.git/commit/043a50fe5a24ce5fbd31d171ae9aeb4d2600accf)
- Project Metadata: Changes to metadata about the data used by the ML pipeline
    (e.g., init, first commit, initial commit, model initial commit)
- Add Dependency: Introduction of a new dependency (e.g., allow flax, create
    requirements.txt)
- Remove Dependency: Removal of an existing dependency (e.g., delete flax, delete
    requirements.txt)
- Update Dependency: Updates to the metadata of an existing dependency (e.g.,
    update checkpoint for transformers>=4.29 (#4))

Please classify the following commit messages:
```

Appendix B: Heuristics for synchronization pattern Labeling

B.1 Algorithm for detecting activity lead between GH and HF

Algorithm 2 Detecting Activity Lead Across Upstream (GH) and Downstream (HF)

Input: Commit data with timestamps and source labels
Output: Activity lead label: `Upstream First`, `Downstream First`, `Simultaneous Activities`, or `No Data`
Remove rows with missing commit dates
if no valid commit dates remain **then**
    return `No Data`
end if
Let `min_date` be the earliest commit date
Assign each commit to a biweekly period:

$$\texttt{biweek} = \left\lfloor \frac{(\texttt{date} - \texttt{min\_date}).\texttt{days}}{14} \right\rfloor + 1$$

Construct a table that maps commit presence to each platform for every biweekly period
for each biweekly period in ascending order **do**
    Let `gh_activity` = True if `GH Commit` has activity
    Let `PTLM_project_activity` = True if any PTLM_project has activity
    if `gh_activity` and `PTLM_project_activity` **then**
        return `Simultaneous Activity`
    else if `gh_activity` and not `PTLM_project_activity` **then**
        return `GH First`
    else if not `gh_activity` and `PTLM_project_activity` **then**
        return `HF First`
    end if
end for
return `No Data`

B.2 Algorithm for detecting overlapping of commit activity between GH and HF

Algorithm 3 Categorizing commit overlap between GH and HF PTLMs

Input: Commit data with timestamps and model names
Output: Overlap type: `CC`, `PC`, `VC`, `AS`, or `No Data`
Remove rows with missing commit dates
if no valid commit dates remain **then**
    return `No Data`
end if
Let `min_date` be the earliest commit date
Assign each commit to a biweekly period:

$$\texttt{biweek} = \left\lfloor \frac{(\texttt{date} - \texttt{min\_date}).\texttt{days}}{14} \right\rfloor + 1$$

Identify periods with GH activity: `github_periods`
Identify all unique PTLM names
Set `all_PTLM_project_overlap_with_GH` to true
for each PTLM **do**
    Get `timeline` for that PTLM
    if `timeline` not fully within `github_periods` **then**
        Set `all_PTLM_project_overlap_with_GH` to false
        break
    end if
end for
if `all_PTLM_project_overlap_with_GH` is true **then**
    return `CC`
end if
Identify all PTLM periods
Compute `overlap_periods` between PTLM and GH periods
if `overlap_periods` is not empty **then**
    return `PC`
end if
For each biweekly period, count unique PTLMs active
Identify periods with at least 3 unique PTLMs
if there are 3 or more such periods and none of them overlap with GH periods **then**
    return `VC`
end if
return `AS`

B.3 Algorithm for detecting the intensity of commit activity between GH and HF

Algorithm 4 Categorizing intensity of coordination between GH and HF PTLM_project

Input: Commit data with timestamps and PTLM_project names
Output: Intensity label: `R`, `F`, `S`, or `No Data`
Remove rows with missing commit dates
if no valid commit dates remain **then**
    return `No Data`
end if
Let `min_date` be the earliest commit date
Assign each commit to a biweekly period:

$$\texttt{biweek} = \left\lfloor \frac{(\texttt{date} - \texttt{min\_date}).\texttt{days}}{14} \right\rfloor + 1$$

Identify periods with GH activity: `github_periods`
Identify all unique PTLM_project names: `PTLM_project`
for each `PTLM_project` in `PTLM_projects` **do**
    Get `timeline_periods` for that PTLM_project
    if the number of `timeline_periods` is greater than 3 **then**
        break
    end if
end for
if all PTLM_project have activity within 3 or fewer biweekly periods **then**
    return `R`
end if
for each `PTLM_project` in `PTLM_projects` **do**
    Get `timeline_periods` for that PTLM_project
    Compute `overlap_periods` = intersection of `PTLM_project_periods` and `github_periods`
    if `overlap_periods` is empty **then**
        continue
    end if
    Initialize `consecutive_count` = 1
    Set `last_overlap` = first element in `overlap_periods`
    for each `current_overlap` in `overlap_periods` starting from second **do**
        if `current_overlap` == `last_overlap` + 1 **then**
            Increment `consecutive_count`
        else
            Reset `consecutive_count` to 1
        end if
        if `consecutive_count` $\geq$ 5 **then**
            return `F`
        end if
        Update `last_overlap` = `current_overlap`
    end for
end for
return `S`